0177

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

DIVISION DE ESTUDIOS DE POSGRADO

FACULTAD DE INGENIERIA

1
2ej

ADAPTABILIDAD EN EL PROBLEMA DE LA RUTA MAS CORTA

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN INGENIERIA

(Investigación de Operaciones)

P R E S E N T A :

MARIA DE LA LUZ GASCA SOTO

TESIS CON
FALLA DE ORIGEN

México, D. F. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

DEDICO ESTA TESIS
A TODAS LAS PERSONAS
QUE HAN CONFIADO EN MI Y
QUE ME HAN ANIMADO PARA
SEGUIR ADELANTE

AL DR. VLADIMIR ESTIVILL CASTRO
POR TODO EL APOYO QUE ME HA
OTORGADO Y POR SU VALOSISIMA
COLABORACION EN EL PRESENTE
TRABAJO.

AGRADEZCO MUY PARTICULARMENTE
SU INCONDICIONAL PACIENCIA.

QUIERO EXTENDER UN MERECIDO
AGRADECIMIENTO A TODAS LAS
PERSONAS QUE DE ALGUNA U DE OTRA
FORMA LABORAN EN EL LABORATORIO
NACIONAL DE INFORMATICA AVANZADA
A. C. Y MUY ESPECIALMENTE A LA
DRA. CRISTINA LOYO V.

GRACIAS POR TODO

NO ES QUE NO PUEDA
HACERLO, SOLO QUE
HAY MUCHAS COSAS
QUE AUN NO HE
APRENDIDO A HACER.

Índice

Lista de Figuras

Lista de Tablas

1 Introducción

Gran parte de los problemas de OPTIMIZACIÓN DISCRETA encuentran un camino hacia su solución cuando se reformulan como problemas de: RUTA MÁS CORTA (RMC) o FLUJO EN REDES. De hecho, los algoritmos para resolver el problema de FLUJO EN REDES utilizan como parte de su solución algoritmos para resolver el problema de la RMC, y, en realidad, la mayoría de los algoritmos para OPTIMIZACIÓN los usan como una *operación o instrucción* más. Esto nos indica que el algoritmo de la RMC es fundamental en el área de OPTIMIZACIÓN DISCRETA [19].

Por lo anterior, el análisis y diseño de algoritmos *eficientes* para el problema de la RMC ha constituido, por más de treinta años[1], en una importante área de investigación en el campo de OPTIMIZACIÓN [2]. Como resultado, el algoritmo de DIJKSTRA ha sido considerado como el *mejor* para resolver el problema de la RMC [19]. En los últimos años, diferentes variantes han propuesto mejoras a su *Desempeño Computacional.*

El *objetivo principal* de este trabajo es mostrar la *Adaptabilidad o No-Adaptabilidad* del algoritmo de DIJKSTRA (y sus variantes) haciendo recomendaciones para su implantación. Concentraremos nuestro análisis en el MODELO COMPUTACIONAL DE COMPARACIONES y estudiaremos diferentes *estructuras de datos* como *implantaciones concretas* del mismo *tipo de datos abstracto, Cola de Prioridades.* Como consecuencia, podremos indicar que variantes del algoritmo de DIJSKTRA, englobadas en el mismo modelo computacional, tienen la *capacidad de aprovechar* la *estructura del ejemplar*[2] del problema. En consecuencia estas variantes adaptivas consumen menos recursos de cómputo, en particular la ejecución es más rápida que en el peor de los casos o en el caso promedio.

Como primer, y básico, acercamiento de nuestro objetivo, haremos uso de una REDUCCIÓN del problema de ORDENAMIENTO a una variante del problema de la RMC. Esta reducción requiere tiempo lineal en el número de datos de entrada, y por conocerse cotas mínimas para el problema de ordenamiento, bajo el modelo de comparaciones, establece la mejor cota mínima conocida para los requerimientos de tiempo, tanto en el caso promedio

[1] Dijkstra presenta su algoritmo para resolver el problema de la ruta más Corta, en 1959 [5]. De manera independiente, en 1960, presentan algoritmos similares Dantsing [4] y Hillierr & Whiting [13].

[2] Estamos usando el término *ejemplar de un problema* como traducción a *instance of a problem* y entenderemos que es la definición de un problema y datos específicos de entrada.

como en el peor de los casos, de todo algoritmo que resuelve el problema de la RMC [16].

Una modificación del algoritmo de DIJKSTRA, que llamaremos DIJKSTRAM, resuelve la variante del problema de la RMC y un análisis del peor caso y del caso promedio establece que el DIJKSTRAM es *óptimo* para ambos casos [16].

Denominaremos DIJKSTRASORT al algoritmo de *Reducción* que resuelve el problema de ordenamiento utilizando el algoritmo DIJKSTRAM que soluciona una variante del problema de la RMC.

El problema de ordenamiento ha sido ampliamente estudiado, en la literatura, *más allá* del tradicional análisis del peor caso y del caso promedio. Existen resultados concretos del problema de ordenamiento sobre sus algoritmos, cotas, implantaciones y adaptabilidad [8].

De esta forma, tenemos una plataforma teórica sólida sobre la cual podemos apoyar y comparar nuestros resultados.

El contenido del presente trabajo se distribuye de la siguiente manera:

- Se inicia con una semblanza general sobre el ANÁLISIS DE ALGORITMOS donde se incluye una introducción a la ADAPTABILIDAD.
- El capítulo II describe el problema de la RMC haciendo énfasis en el algoritmo de DIJSKTRA.
- La tercera parte, describe un marco teórico del planteamiento central de la tesis y la manera como será desarrollada su solución.
- El capítulo IV presenta el algoritmo DIJKSTRASORT.
- En los capítulos V y VI se realizan ANÁLISIS DE ADAPTABILIDAD de los algoritmos DIJKSTRASORT y DIJKSTRA, para diversas implantaciones y diferentes tipos de gráficas.

Los *apéndices* que complementan este trabajo son:

- *A.* Se describen detalladamente los diferentes métodos para resolver la RMC.
- *B.* Se describen diferentes Tipos de Datos Concretos del Tipo de Datos Abstracto Cola de Prioridades.
- *C.* Se introducen conceptos básicos de la Programación Orientada a Objetos.
- *D.* Se introducen conceptos básicos sobre adaptabilidad y medidas del desorden, además se presentan resultados concretos del análisis de adaptabilidad en el problema de ordenamiento.
- *E.* Se presentan ideas para el análisis de adaptabilidad del Algoritmo de Dijkstra usando Radix Heaps.

I Análisis de Algoritmos

Para aprovechar al máximo el los recursos que ofrece la tecnología de las máquinas actuales es indispensable el desarrollo de algoritmos que tomen ventaja de tales recursos. El campo de las Ciencias de la Computación que estudia la eficiencia de algoritmos es conocido como Análisis de Algoritmos [16].

En esta sección introduciremos conceptos y notación sobre el análisis y adaptabilidad de algoritmos, los cuales serán utilizados durante el desarrollo de este trabajo.

1 Panorámica General

Antes de iniciar, con una semblanza general sobre el análisis de algoritmos, definiremos formalmente algunos conceptos básicos.

- ⋆ **Un problema** es una cuestión a resolver, generalmente, posee varios parámetros. Por ejemplo, encontrar el máximo en un conjunto finito de números naturales.
- ⋆ **Un ejemplar de un problema** *(instance of a problem)* es es una asignación de valores para los parámetros. Por ejemplo, encontrar el máximo en $\{ 3, 7, 5, 8, 1 \}$.
- ⋆ **Un algoritmo** para un problema es una descripción del procedimiento paso a paso para la obtención de la solución del problema. Un Algoritmo es tal que, al tomar un ejemplar cualquiera del problema produce la respuesta correcta para tal ejemplar. Si muchas respuestas son equivalentementemente correctas el algoritmo será capaz de producir cualquiera de ellas o generarlas todas.
- ⋆ **Un algoritmo es correcto** si garantiza la creación de una respuesta correcta para cada ejemplar del problema.
- ⋆ **Un ejemplar de un problema de optimización** es una pareja (F, c), donde F es cualquier conjunto, el dominio de puntos *factibles*, y c es la *función de costo*, $c : F \longrightarrow \mathbb{R}$. El problema es encontrar $f \in F$ para la cual $c(f) \leq c(y)\ \forall\, y \in F$, a f se le llama *solución óptima global* para el ejemplar dado.
- ⋆ **Un problema de optimización** es un conjunto E de ejemplares de un problema de optimización.
- ⋆ **Un algoritmo eficiente** es aquel para el cual el tiempo de ejecución, en el peor de los casos, está acotado por una función polinomial que depende del tamaño del ejemplar del problema.

⋆ Informalmente, un **ejemplar** es el conjunto de datos suficientes para obtener alguna solución de un problema. Y un **problema** es un conjunto de ejemplares.

Al intentar resolver un problema se podrían tener varios algoritmos que los solucionen, entonces surgen algunas preguntas como:

- ¿Cuál es el correcto?, ¿Cuál es el mejor?, ¿Cuál es el más rápido?
- ¿Cuál es el que gasta menos espacio de almacenamiento?
- ¿Cuál es el más fácil de ...
  - ... entender?
  - ... programar?
  - ... implantar?
  - ... adaptar a los sistemas ya programados?
  - ... modificar?
  - ... aplicar a más casos?

Pero muy pocas veces se hace referencia a preguntas como:

- ¿qué tan rápido, eficiente y eficaz resulta ser un algoritmo para un problema en particular?;
- ¿qué tan bueno es para un tipo específico de máquina?;
- ¿qué tan eficiente resulta ser un algoritmo si se implanta en un lenguaje de programación en particular?;
- ¿qué tan eficiente resulta ser si se utiliza un compilador en particular?;

porque su respuesta es demasiado específica.

Generalmente, se intenta responder a las preguntas del tipo:

- ¿ Qué tan rápido es el *Algoritmo* ... para solucionar el *Problema* ... ?

Y se obtienen respuestas que no dependen ni de la máquina ni del problema.

De esta manera, *Udi Manber* [21] afirma que:

> " ... el propósito del análisis de algoritmos es predecir el comportamiento de un algoritmo sin implantarlo en una máquina específica ... "

Para lograr independencia sobre el tipo de computadora, concebimos a ésta como una *simple máquina de operaciones elementales*. Luego, si un algoritmo usa menos operaciones elementales que otro será más rápido. Al cálculo de operaciones elementales se le llama **tiempo de ejecución del algoritmo.**

Una importante y práctica razón para justificar el análisis de algoritmos es descrita por Goodman [10]

" ... se requiere obtener estimaciones o cotas sobre el almacenamiento y el tiempo que toma en ejecutarse el algoritmo, ya que la memoria y el tiempo de cálculo de la máquina son recursos, generalmente, escasos y, sobre todo, costosos ... "

Así pues, el análisis y comparación de algoritmos, tradicionalmente, se ha basado en el tiempo de ejecución y tamaño de memoria utilizado para almacenar los datos sobre todos los ejemplares del problema.

Para estudiar la eficiencia de algoritmos necesitamos un modelo computacional [45]. Un **Modelo Computacional** es un conjunto de suposiciones que se asumen acerca de la máquina (virtual o real) sobre la cual será ejecutado el algoritmo. En el presente trabajo, usaremos el *Modelo Computacional de Comparaciones* . Este modelo, divide un ejemplar E en dos partes: una libre, f , y otra restringida r ; de esta forma, $E =< f, r >$. Se permite que un algoritmo use f de la forma en que lo desee, pero r únicamente podrá usarse en operaciones de comparación. [16, 45]

Tarjan[45], nos indica que una vez elegido el modelo computacional, se debe seleccionar una medida de desempeño. Tomaremos al *tiempo de ejecución* como medida de desempeño. Formalmente, el **tiempo de ejecución** de un algoritmo A, aplicado a un ejemplar E, es el número de pasos a efectuar en A para encontrar la solución para E , y lo denotaremos por $T_A(E)$, en particular, el valor de $T_A(E)$ reoresenta al menos el número de comparaciones realizadas para encontrar la solución del ejemplar E.

El análisis de un algoritmo A en todos lo casos, implicaría decir para cada ejemplar E de un problema, el tiempo de ejecución $T_A(E)$ que toma A en resolver E. Como usualmente el conjunto de todos los ejemplares E es muy grande, sólo trabajaremos con familias de ejemplares de un mismo tamaño. Un ejemplar E es de tamaño n si tiene n datos o alguna medida natural del tamaño, y se denota por $|E|$. De esta forma, más adelande definiremos $T_A(n)$ como el desempeño para la clase de ejemplares de tamaño n . Más aún, sólo nos interesa el orden de magnitud de $T_A(n)$, pues la implantación particular se afecta únicamente por un factor constante.

Si $T(n)$ está asintoticamente acotada por $g(n)$ decimos que $T(n)$ es de orden $g(n)$ y lo denotamos por $T(n)$ es $O(g(n))$. Más formalmente, Si f y g son funciones de variables no negativas $n_1, n_2, ..., n_k$ se dice que:

▷ f es $O(g)$ si $\exists\, c_1, c_2$ constantes positivas tales que

$$f(n_1,...,n_k) \leq c_1 \cdot g(n_1,n_2,...,n_k) + c_2 \quad \forall\, n_i$$

{ representa una cota superior para f }

▷ f es $\Omega(g)$ si g es $O(f)$. { representa una cota inferior para f }

▷ f es $\Theta(g)$ si f es $O(g)$ y $\Omega(g)$.

El desempeño computacional de un algoritmo puede ser analizado como una función del *peor de los casos*, del *caso promedio* o *amortizada* [10, 9, 16, 45, 43].

Como una función para el peor de los casos, $T_A(n)$ se define de la siguiente forma:

$$T_A(n) = T_A(E^*) \quad \text{donde} \quad T_A(E^*) = \max\{T_A(E) \;/\; E \text{ es de tamaño } n\}.$$

El análisis del peor de los casos nos proporciona una ejecución garantizada del algoritmo. Esto es, el algoritmo nunca requerirá más tiempo (o espacio) de que ha sido especificado por la cota dada. Resulta ser muy pesimista.

El análisis del caso promedio, nos proporciona una función que depende de la esperanza de $T_A(n)$, esto es $\mathbf{E}[T_A(n)] = \sum_{\forall\, |E|=n} \mathbf{Prob}[E] \cdot T_A(E)$, donde $\mathbf{Prob}[E]$ es la probabilidad de que ocurra E.

El análisis del tiempo amortizado es un promedio sobre el tiempo [44]; esto es, se toma $T_A(E)$ para diferentes instantes sobre la ejecucion del algoritmo y después se calcula el promedio sobre esos valores. Sea $T_A(E)_i = T_A(E)$ en el instante i. Si tomamos $T_A(E)$ para η instantes diferentes, entonces el tiempo amortizado queda definido por:

$$T_A(n) = \frac{1}{\eta} \cdot \sum_{i=1}^{\eta} T_A(E)_i$$

Ventajas de los Tipos Análisis de Algoritmos

- Analizar un algoritmo permite observar *varias alternativas* de desarrollo e implantación del mismo o partes de él.
- Resulta *más conveniente* establecer medidas simples para evaluar la eficiencia de un algoritmo que implantarlo y probar que tan eficiente es cada vez que algún parámetro básico de la computadora cambie.

2 Adaptabilidad

El análisis del peor de los casos, del caso promedio y del tiempo amortizado, nos proporcionan información sobre las virtudes de un algoritmo y nos ayudan a elegir entre varios de ellos.

Actualmente ha surguido otra panorámica para proporcionar mayor información, evaluar, analizar y diseñar algoritmos, denominada **adaptabilidad de algoritmos:**

> Se dice que un **algoritmo es adaptivo** [3], si para *cada ejemplar dado*, la realización de operaciones elementales resulta ser una función no decreciente que depende del tamaño y dificultad del ejemplar del problema [7].

Informalmente, diremos que un algoritmo es adaptivo si es capaz de aprovechar las características del ejemplar del problema a solucionar. Más formalmente, un algoritmo es adaptivo si la información sobre $T_A(E)$ no sólo se expresa como $T_A(|E|)$, sino como una función $T_A(|E|, dif(E))$, donde $dif(E)$ es una función que mide la dificultad del ejemplar[8].

En general, la mayoría de los algorimos son glotones, esto es, utilizan o barren toda la estructura del ejemplar para intentar solucionarlo. Se tiene, entonces, que la *adaptabilidad* es una propiedad que poseen algunos algoritmos.

En este trabajo deseamos mostrar si existe adaptabilidad en algunas implantaciones de los algoritmo de LA TEORÍA DE REDES, en especial los que resuelven el problema de la RMC y más específicamente del algoritmo de DIJKSTRA.

Adaptabilidad en Redes

> Diremos, de manera informal, que un **algoritmo en redes es adaptivo** si es capaz de ajustarse o amoldarse, por sí mismo, a la estructura específica de la red, disminuyendo el desempeño computacional del algoritmo para tal ejemplar.

[3] Nos referiremos a *algoritmo adaptivo* como traducción de *adaptive algorithm*.

II Ruta Más Corta

Esta sección describiremos el problema de la Ruta Más Corta, presentaremos, de manera general, modelos y algoritmos que resuelven este problema, haciendo énfasis en el algoritmo de DIJKSTRA.

1 El Problema de la Ruta Más Corta

RMC: Dada una red G con n vértices, el problema consiste en encontrar la RUTA MÁS CORTA (longitud) o más barata (costos) entre dos o más vértices.

El Problema RMC es básico para los problemas de OPTIMIZACIÓN COMBINATORIA y FLUJO EN REDES. Algoritmos que resuelven estos problemas lo utilizan como una operación más que los complementa [18]. Este tipo de problemas tiene aplicaciones en *modelos de distribución* y *asignación de recursos*, entre otras aplicaciones de la TEORÍA DE REDES. Su aplicación más importante sucede cuando se combinan problemas de *flujo máximo* en redes a *costo mínimo*, esto es el problema de *Flujo Máximo a Costo Mínimo* en una red [36].

1.1 Modelos para la Ruta Más Corta

El problema de RMC puede agruparse en los siguientes modelos:

i) Encontrar la RMC desde un vértice a todos los demás vértices de G cuando las longitudes de los arcos son *no negativas*.

ii) Encontrar la RMC desde un nodo x a todos los otros nodos de la *Red* G *sin importar* si las longitudes de los arcos son positivas o no.

iii) Encontrar la RMC entre todo par de vértices.

iv) Otras generalizaciones de la RMC.

Para resolver **(i)** y **(ii)** se cuenta con algoritmos que consisten de métodos de etiquetación cuando las longitudes de los arcos son no negativas y métodos de corrección de etiquetas para el caso en que las longitudes pueden ser negativas. En este trabajo nos concentraremos en resolver **(i)**.

1.2 Algoritmos para la Ruta Más Corta

Los algoritmos para RMC se pueden clasificar, según su construcción, básicamente en dos tipos:

- ⋆ *Algoritmos que construyen Arboles de* RMC

  los cuales encuentran la RMC del origen, s, a cada uno de los otros vértices.
  (RMC entre dos vértices específicos)

- ⋆ *Algoritmos auxiliados por Matrices*

  que localizan la RMC entre cada par de vértices.
  (RMC entre todo par de vértices)

Los algoritmos de RMC entre dos vértices específicos basan su *solución analítica* en las **Ecuaciones de Bellman**. La multiplicación matricial constituye una herramienta fundamental para los algoritmos que resuelven la RMC entre todo par de vértices. La descripción detallada de estos algoritmos se encuentra en el *Apéndice A*, [14, 46, 18, 19].

1.3 Notación

En el presente trabajo denotaremos por:

$\mathbf{G} = \langle \mathbf{N}, \mathbf{A}, \delta \rangle$ a una red con costos en los arcos, donde
$\mathbf{N}$ es el conjunto, no vacío y finito, de nodos o vértices;
$\mathbf{A}$ es el conjunto de arcos o aristas $\mathbf{A} \subseteq \mathbf{N} \times \mathbf{N}$; y
$\delta : \mathbf{A} \longrightarrow \mathbb{R}^+$ es la función de costo sobre los arcos.

$n = |\mathbf{N}|$ es el número de nodos o vértices.

$m = |\mathbf{A}|$ es el número de aristas o arcos.

(i, j) es el arco de i a j.

a_{ij} es la longitud o costo del arco (i,j).

s, d representan a los nodos origen y destino, respectivamente.

$d(v)$ es la etiqueta de v.

$\mathcal{P}_{vw}$ representa una ruta de v a w; para toda $v, w \in \mathbf{N}$.

$\mathbf{RMC}(s, w)$ representa la ruta más corta entre los vértices s y w.

$\mathcal{A}(s) = \langle N, A', \delta_{|A'} \rangle$ es una arborescencia de G donde $A' \subseteq A$ y $\delta_{|A'}$ es la función de costo δ restringida a A'.

$\mathcal{A}^*(s)$ es una Arborescencia $\mathcal{A}(s)$ de G para la cual cada ruta $\mathcal{P}_{sx}$ es única y representa la RMC(s, x), para toda $x \in A$.

La Longitud o Costo de una ruta es la suma de las longitudes o costos de los arcos que la constituyen.

2 Algoritmo de Dijkstra

El Algoritmo de DIJKSTRA es uno de los más eficientes para resolver el Problema de RMC, de un nodo fuente s a cada uno de los otros nodos en una *Red* G para la cual las longitudes de los arcos, a_{ij}, son no negativas. Es considerado el algoritmo básico de etiquetamiento [19].

En esta sección daremos una panóramica general del algoritmo, así como una justificación matemática que nos garantiza que el algoritmo es correcto.

2.1 Panorámica General del Algoritmo

- Cada nodo i posee una etiqueta $d(i)$ donde:
  - $d(i)$ es **permanente** si representa la RMC de s a i.
  - $d(i)$ es **temporal** si aún no representa la RMC de s a i; es decir, está por encima de la mínima longitud posible.
- El algoritmo inicia etiquetando los vértices:
  - El origen s, será permanente con $d(s) = 0$
  - Los demás tendrán etiqueta temporal, dada por:

$$d(j)_{j \in N \setminus \{s\}} = \begin{cases} a_{sj} & \text{si } \exists\ (s,j) \in A; \\ \infty & \text{si } \nexists\ (s,j) \in A. \end{cases}$$

- Se busca entre los nodos con etiqueta *temporal* al nodo k cuya etiqueta, $d(k)$, resulta ser la mínima.

  Si $d(k) \neq \infty$
  - Tal $d(k)$ se convierte en permanente.

▷ Se revisan las etiquetas temporales de los nodos j adyacentes al vértice k, actualizando de la siguiente forma sus etiquetas:

$$d(j) \longleftarrow \min_{\forall\, j \text{ ady a } k} \{ d(j),\ d(k) + a_{kj} \}$$

▷ El proceso termina cuando todos los nodos han sido etiquetados de manera permanente o cuando todas las etiquetas. $d(k)$, tienen como valor mínimo a infinito.

El algoritmo queda determinado, en forma general, de la siguiente manera:

ALGORITMO DE DIJKSTRA

I Definición inicial de etiquetas.

II Mientras existan Etiquetas Temporales Finitas

1. Selección del nodo, k, con etiqueta temporal mínima
2. Si $d(k)$ es finita
   a) $d(k)$ se convierte en etiqueta permanente
   b) Actualización de las etiquetas temporales de los nodos adyacentes a k.

2.2 Justificación del Algoritmo

Resulta claro que el algoritmo es finito, ya que existe un número finito de vertices, por tanto hay un número finito de etiquetas temporales y por consecuencia se tiene un número finito de etiquetas temporales con valor finito. Esto es:

$$\{ \text{Etiquetas temporales con valor finito} \} \subseteq \{ \text{Etiquetas temporales} \} \subseteq \mathbf{N} \setminus \{s\} \subset \mathbf{N}.$$

Observamos que el algoritmo termina cuando ocurre una de las siguientes condiciones:

1. Todas las etiqueta han sido clasificadas como permanentes. Mostraremos que entonces existe una arborescencia $A^*(s)$, de rutas más cortas.
2. Ya no hay etiquetas temporales con valor finito, pero aún existen etiquetas temporales. Demostraremos entonces que no existe una arborescencia $A^*(s)$.

Caso 1. Supongamos que el algoritmo termina cuando todas las etiquetas han sido marcadas como permanentes. Por demostrar que la arborescencia, $A^*(s)$, encontrada es de rutas más cortas. Demostraremos por inducción que después de n iteraciones del paso II, para cada vértice que

A) Si v tiene etiqueta permanente, $d(v)$, entonces ésta representa la RMC(s, v).

B) Si un vértice w tiene etiqueta temporal, $d(w)$ es la longitud de una ruta $\mathcal{P}_{sw}$ para la cual, los vértices que la forman tiene etiquetas permanentes, excepto w.

Base de la Inducción. Sea $n = 1$.

Después de realizar el paso I: la etiqueta $d(s) = 0$ y es permanente. La etiqueta $d(j)$ es temporal para toda $j \in N\backslash\{s\}$, además $d(j) = a_{sj}$ siempre que $\exists (s, j) \in A$.

Para el paso II.

Si $N\backslash\{s\} = \emptyset$, el algoritmo termina, pues ya no hay etiquetas temporales. La RMC(s, s) tiene longitud $d(s)$. □

Si $N\backslash\{s\} \neq \emptyset$, aún hay etiquetas temporales. Hasta el momento, sólo hay una etiqueta permanente, $d(s) = 0$ es claro que $d(s)$ reprsenta la RMC(s, s). □

Ahora bien, para cualquier vértice ω con etiqueta temporal finita, tenemos que $d(\omega) = a_{s\omega}$ y la ruta $\mathcal{P}_{s\omega}$ está formada únicamente por dos vértices, s y ω, donde s es permanente y ω no. □

Paso de Inducción.

Supongamos que para $n = k$ las afirmaciones (A) y (B) son válidas. Por demostrar que también son válidas para $n = k + 1$.

Sea v el vértice que acaba de ser elegido como mínimo, entonces

$$d(v) = \min_{d(j)\ temporal} \{d(j)\} \quad \text{y} \quad d(v) \text{ es la nueva etiqueta permanente.}$$

Por demostrar que $d(v)$ representa la RMC(s, v).

Supongamos que $d(v)$ no representa la RMC(s, v).

Sea $\mathcal{P}_{sv}$ la ruta de s a v con longitud $d(v)$. Como $\mathcal{P}_{sv}$ no representa la ruta de s a v con mínima longitud, entonces existe una ruta, digamos $\mathcal{R}_{sv}$, para la cual la longitud de $\mathcal{R}_{sv}$ es menor que $d(v)$, Figura 1. Esta ruta debe contener al menos un vértice con etiqueta temporal finita. Sea δ el primer vértice apartir de s con etiqueta temporal finita. Entonces la longitud de la subruta de s a δ, $\mathcal{R}'_{s\delta}$, debe ser menor que la longitud de $\mathcal{R}_{sv}$. Pero la subruta $\mathcal{R}'_{s\delta}$ sólo contiene un vertice temporal finito, por lo tanto: $d(\delta) \leq d(v)$. Esto contradice la elección, δ debio ser elegido antes que v, por lo tanto tal $\mathcal{R}_{sv}$ no existe.

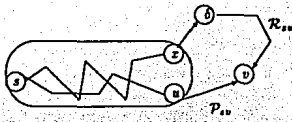


Figura 1: Rutas más cortas de s a v

Por demostrar, ahora, que para todos los vértices w con etiqueta temporal finita, $d(\omega)$ representa la longitud de una ruta $\mathcal{P}_{sw}$ en la cual todo vértice tiene etiqueta permanente, excepto ω.

Supongamos que tal proposición es falsa. Esto es, existe un vértice w con etiqueta temporal finita y una ruta $\mathcal{P}_{sw}$ de longitud r, en la cual w es el único vértice con etiqueta temporal finita, donde además $r < d(w)$.

Sea x un nodo con etiqueta permanente, adyacente a w, en la ruta $\mathcal{P}_{sw}$ y sea v es nodo que estamos revisando. Si $x \neq v$ entonces, por hipotesis de inducción $d(w)$ fue determinada (hecha permanente) después de la k-ésima iteración, entonces $r = d(w)$, lo cual contradice el hecho de que $r < d(w)$.

Si $x = v$, entonces $d(w) = d(v) + a_{vw} = r$. Esto contradice el hecho de que $r < d(w)$. Por tanto r no puede ser menor que $d(w)$, entonces la proposición es válida para $n = k+1$.

Caso 2. Supongamos que el algoritmo termina cuando ya no hay etiquetas temporales con valor finito, pero aún existen etiquetas temporales. Por demostrar que no existe una arborescencia, con raíz s, de rutas más cortas.

Tenemos que existe al menos un vértice x para el cual $d(x) = \infty$, esto nos indica que *no* existe una ruta que una a s con x, esto es: $\exists\, x \in \mathbf{N} \;.\!\!+\!\!.\; \nexists\, \mathcal{P}_{sx}$. Entonces, no existe una arborescencia para G, por lo tanto, no existe una arborescencia, $\mathcal{A}^*(s)$, de rutas más cortas con raíz s.

Con lo cual queda demostrado que el algoritmo es correcto.

III Marco Teórico

En este capítulo iniciamos presentando formalmente el concepto de *Tipo de Datos Abstracto (TDA)*, especificando el *TDA Cola de Prioridades* e introduciendo el *Paradigma Orientado a Objetos*. Después se plantea un *Marco Teórico*, con los conceptos definidos, que nos permitirá desarrollar los objetivos de este trabajo. Finalmente, se describen los principales *TDA* para el Algoritmo de DIJKSTRA y se describe éste en términos de las operaciones de tales *tipos de datos*.

1 Tipos de Datos Abstractos

El desempeño computacional de un algoritmo dependerá de la forma como se representen y manipulen los datos de entrada (ejemplares) y las estructuras auxiliares que permiten la realización del mismo. A la forma específica de representar datos iniciales, temporales, auxiliares y finales, le llamaremos **Estructuras de Datos**. En particular, el Algoritmo de DIJSKTRA basa su desempeño computacional en la forma como se reprenta, almacena y opera su Ejemplar, la Red y las Etiquetas Temporales.

Las estructuras de datos, como nos indica J.J. Martin [15], sirven para implantar datos (objetos) complejos, los cuales se clasifican en *tipos* de acuerdo a la manera como serán usados. Así que, dependiendo del punto de vista, un dato puede ser caracterizado por su tipo (para su uso) o por su estructura (para su implantación).

La **Abstracción de Datos** consiste en separar las estructuras de datos del algoritmo, lo cual nos permite el estudio de cada estructura por separado y nos provee una manera de organizarlas y simplificarlas. De esta forma contamos, con una gran variedad de medios que no dependen de un algoritmo en particular para representar y estructurar datos, pero que poseen operaciones sobre tales estructuras. La *unión* de un conjunto de estructuras y las operaciones que las manipulan se ven englobadas en una *Entidad Matemática* [16].

Desde una panorámica más abstracta, sabemos que el concepto de entidad matemática no obliga a una representación específica y posee una serie de axiomas y operaciones particulares (ejemplos: conjuntos, espacios vectoriales).

Un **Tipo de Datos Abstracto** es, básicamente un conjunto de objetos y un conjunto

de operaciones que manipulan tales objetos [15]. De manera similar J.H. Kingston [16], define, Tipo de Datos Abstracto (TDA) como una entidad matemática con operaciones propias definidas sobre la entidad: $\ll$ *adt* , *operaciones* $\gg$. De esta forma, los objetos manipulados por un algoritmo se conciben como objetos matemáticos a través del concepto de Tipo de Datos Abstracto.

La importancia de este concepto ha sido reconocido muy recientemente, aunque se ha usado desde el inicio de la computación.

J.J.Martin [15], nos dice que:

> Crear e implantar nuevos *tipos de datos* se ha convertido en una fuerte herramienta para separar niveles clave en programas grandes. Para explotar esta herramienta, se debe aprender a
>
> - reconocer los *objetos* candidatos a formar nuevos *tipos de datos*;
> - identificar las *operaciones básicas* para tales *objetos*;
> - especificar con precisión las *operaciones básicas*; y
> - seleccionar una *buena implantación*.

La mejor opción de implantación de un TDA es la *Programación Orientada a Objetos (POO)*. La cual, ha venido a ser consecuencia de la evolución de los lenguajes de programación. B. Eckel [6], nos da una buena panorámica de la evolución de los lenguajes de programación:

> Usando el *lenguaje ensamblador*, el programador podía evitar la codificación de números y, en su lugar, pensar en palabras. Los *lenguajes procedurales* escondían la complejidad de las operaciones sobre los datos. Los *lenguajes orientados a objetos (LOO)* esconden la complejidad del programa en sí mismos.
>
> Un *LOO* enfatiza los *tipos de datos y sus operaciones*. En la *POO*, los datos no fluyen abiertamente alrededor del sistema, son protegidos de modificaciones accidentales. Los *Mensajes*, en vez de los datos, son los que se mueven en el sistema. Ya no se tiene un acceso procedural, en los *LOO* se manda un *mensaje* a un *objeto*.

En la *POO* , los *objetos* y sus *operaciones básicas* se describen conjuntamente. A las *operaciones* se les denomina **Mensajes** o **Caracteristicas**[4]. A la definición conjunta de un objeto y sus caracteristicas se le llama **Clase**. Las caracteristicas van asociadas al

[4]En este trabajo las llamaremos *Caracteristicas*, pues usaremos la tecnología *OO* de Eiffel [24]

objeto. Por ejemplo, si un objeto **c** de la clase *Números Complejos* tiene, en particular dos características: la parte *Real* y la parte *Imaginaria* la forma de invocar a la parte real es **c.***Real*.

Denominaremos por *Tipos de Datos Concretos (TDC)* a la definición concreta de cualquier *TDA*. Los *LOO* son la tecnología más avanzada para dar una especificación de los *TDC* mediante definiciones de *clases*.

1.1 Colas de Prioridades

Una **Cola de Prioridades** es un TDA con muchísimas aplicaciones. Su nombre proviene de una aplicación en sistemas operativos: el mantenimiento de una cola de procesos los cuales serán ejecutados según su prioridad; en realidad éstas son usadas diariamente en varias actividades cotidianas, tales como *La sala de espera de un hospital*, donde cada paciente será atendido según la gravedad de su caso.

Recordemos que en el Algoritmo de DIJKSTRA, nos interesa recuperar la etiqueta temporal mínima, así pues requerimos manipular un conjunto mediante operaciones que faciliten cosas como, agregar un elemento, encontrar y borrar el mínimo elemento.

Especificaciones

Una **Cola de Prioridades de tipo T** con prioridades reales, es un TDA cuyos objetos son los conjuntos **Q** de objetos en T. Cada uno de los elementos en **Q** tiene asociado un número real, k, al cual se le denomina *llave del objeto*. Las operaciones válidas sobre un conjunto $\mathbf{Q} \in 2^T$ son las siguientes:

Inserta (x :**objeto**, k :**llave**) .- Incluye un nuevo objeto x, con llave k a la colección de objetos **Q**.

EncuentraMin : objeto .- Encuentra y regresa el objeto cuya llave es mínima en la colección **Q**.

BorraMin (x :**objeto**) .- Borra un objeto x, cuya llave es mínima en la colección de objetos **Q**.

Inicia .- Crea una nueva estructura de tipo *Cola de Prioridades*.

Vacio .- Indica si la colección de objetos **Q** es o no vacía.

Elimina (x :**objeto**) .- Elimina un objeto x de la colección de objetos **Q**.

DecrementaLlave(x :objeto, k :Llave) .- Toma al objeto x de **Q** y cambia el valor de su llave a k, sólo si k es menor que el valor de la llave anterior.

Formalmente, una cola de prioridades es una n-ada de objetos de tipo T y prioridades en los números reales, $\mathbb{R}$. Esto es: $2^T \times (T \longrightarrow \mathbb{R})$, donde:

$A \times B$ es el producto cartesiano de A con B;

2^T es el conjunto potencia de T;

$T \longrightarrow \mathbb{R}$ es el conjunto de funciones de T en $\mathbb{R}$.

Por ejemplo, la operación **Inserta** queda formalmente definida, de la siguiente manera:

$$\textbf{Inserta} : [\,2^T \times (T \rightarrow \mathbf{R})\,] \times [\,T \times \mathbb{R}\,] \longrightarrow [\,2^T \times (T \rightarrow \mathbb{R})\,]$$

donde.

Si $\mathbf{Q} \in 2^T$, $\mathbf{pr} \in T \longrightarrow \mathbb{R}$, $x \in T$. y $k \in \mathbb{R}$ entonces,

Si $(\mathbf{Q}', \mathbf{pr}') = \textbf{Inserta}((\mathbf{Q}, \mathbf{pr}), (x, k))$, $x \in \mathbf{Q}'$ y $\mathbf{pr}'(x) = k$.

Un ejemplo de *axioma*:

Si la llave k del nuevo elemento insertado x es menor a todas las que ya había en **Q**, entonces x es el nuevo elemento mínimo de la cola **Q**. Esto es:

Si $\forall\, y \in \mathbf{Q}$, $\mathbf{pr}(y) \geq k \implies \textbf{EncuentraMin}(\textbf{Inserta}((\mathbf{Q}, \mathbf{pr}), k)) = x$.

2 Descripción del Marco Teórico

A mediados de la década de los 70's, Wirth [48] sugiere una *relación* entre los *algoritmos*, las *estructuras de datos* y los *programas*:

$$\textit{Algoritmos} + \textit{Estructuras de Datos} = \textit{Programas}. \qquad \ldots\ \mathcal{R}$$

Para el presente trabajo, esta *relación* nos proporciona una interesante forma de ver las diferentes versiones o implantaciones de un algoritmo en un programa.

Si fijamos un Algoritmo A y variamos las estructuras de datos, tendremos, seguramente, diferentes programas, y en consecuencia diferentes desempeños y tiempos de ejecución.

Si en la relación $\mathcal{R}$ *fijamos* el algoritmo de DIJKSTRA, tenemos que

$$\begin{bmatrix} Algoritmo \\ de \\ Dijkstra \end{bmatrix} + \begin{bmatrix} Estructuras \\ de \\ Datos \end{bmatrix} = [\ Programas\ ]$$

Tomando en cuenta que, las estructuras de datos pueden ser englobadas en un *TDA*, podemos considerar una *conceptualización abstracta* del algoritmo, la cual no depende de ningún *TDC*, esto es:

$$\begin{bmatrix} Algoritmo \\ de \\ Dijkstra \end{bmatrix} + \begin{bmatrix} Tipos \\ de\ Datos \\ Abstractos \end{bmatrix} = \begin{bmatrix} Conceptualización \\ Abstracta \end{bmatrix}$$

Utilizando la *POO*, los *Tipos de Datos Abstractos* se convierten ahora en *Objetos y Clases*:

$$\begin{bmatrix} Algoritmo \\ de \\ Dijskta \end{bmatrix} + \begin{bmatrix} Objetos \\ y \\ Clases \end{bmatrix} = \begin{bmatrix} Implantación \\ Orientada \\ a\ Objetos \end{bmatrix}$$

Ahora bien, la *especificación concreta* de los *objetos* y las *Clases* nos determinará un *Sistema*:

$$\begin{bmatrix} Algoritmo \\ de \\ Dijskta \end{bmatrix} + \begin{bmatrix} Especificación\ Concreta \\ de \\ Objetos\ y\ Clases \end{bmatrix} = [\ Sistema\ ]$$

La siguiente tabla, nos muestra los Desempeños Computacionales, conocidos en la literatura para el peor caso, para diferentes versiones del algoritmo DIJKSTRA [1, 2].

Tipos de Datos Concretos		
RED	Conjunto de ETIQUETAS	Desempeño Computacional
Listas de Adyacencia	Listas Ligadas	$O(n^2)$
	Heap-Binarios	$O(m \cdot log_2 n)$
	d-Heap	$O(m \cdot log_d n)$
	Fibonacci-Heaps	$O(m + n \cdot log_2 n)$
	Radix-Heaps	$O(m + n \cdot log_2(nC))$
Matriz de Incidencia	Cualquiera de las anteriores	$O(n \cdot m)$
Matriz de Adyacencia	Cualquiera de las anteriores	$O(n^2)$

Para el presente trabajo, describiremos una *Conceptualización Abstracta* del algoritmo de DIJKSTRA, utilizando notación de *POO*, sobre la cual calcularemos el Desempeño Computacional para el peor de los casos. Después iremos especificando los *TDA*. Para cada implantación concreta *refinaremos*, de ser posible, el desempeño computacional, tanto para el peor de los casos como para el caso promedio, y finalmente, realizaremos el análisis de adaptabilidad.

3 TDA's en el Algoritmo de Dijkstra

Al intentar implantar un algoritmo, debemos decidir la mejor representación de los datos y surge, entonces, el balance de factores no todos alcanzables simultaneamente. Analicemos un poco al algoritmo de DIJKSTRA, sus ejemplares y demás estructuras auxiliares, necesarias para llevar a cabo su ejecución. Un ejemplar para el problema de la RMC, es una red con costos en los arcos. En particular, el Algoritmo de DIJKSTRA, se efectúa eficientemente sobre una red con costos positivos.

TDA para la RED G

Representaremos a la red G como una Lista de Adyacencias, las operaciones básicas requeridas para identificar el Tipo de Datos Abstracto *RED*, son:

CreaRed.- Genera un objeto de tipo RED.
AccesaAdyacentes (*v*: Vértice) .- Regresa la lista de los vértices adyacentes a *v*.
GuardaArco (*a*: Arco; *c*: Tipo_Costo) .- Añade el arco *a* con costo *c* en la red.
PrimerVertice.- Regresa el primer vertice de la red.
PrimerArco.- Regresa el primer arco en la red.
VerticeFinal (*a*: Arco) .- Regresa el vertice final del arco *a*.
ImprimeRed .- Muestra los datos contenidos en la red.

TDA para las ETIQUETAS TEMPORALES

Las *etiquetas temporales* resultan ser una estructura auxiliar de almacenamiento fundamental en el desarrollo del Algoritmo de DIJKSTRA, por ello la forma como se representen será crucial para el cálculo del desempeño computacional. Las etiquetas temporales tienen operaciones específicas que dan prioridad a las etiquetas con el mínimo valor. Entonces, podemos englobarlas en el *TDA* definido anteriormente como *Colas de Prioridades*, al cual denominaremos *PQ*.

TDA para la RUTA

Durante la ejecución del algoritmo de DIJKSTRA, cada vértice de la red pasa de la situación de *no-revisado* a *ya-revisado*: el valor de su etiqueta puede ir disminuyendo. En el cálculo de rutas mínimas el predecesor de un vértice en la arborescencia puede cambiar. Por otra parte, queremos tener acceso directo a la etiqueta del vértice en la cola de prioridades, pues teniendo esta información se puede 'reconstruir' la RMC de un vértice a otro, o, bien, se puede 'reconstruir' la arborescencia de RMC. Por esto, para cada vértice en la red **G** declaramos un *Estado* que almacene tal información. las operaciones básicas para el TDA *ESTADO* son:

Visit.- Característica que indica si el vértice ha sido o no visitado.
Padre.- Característica que indica cuál es el vértice predecesor en la Arborescencia.
Dist.- Característica que indica cuál es el valor de la RMC encontrada.
CreaEstado.- Crea un objeto de tipo *ESTADO*.
ActualizaVisit (*vi*: Lógico) .- Modifica la caracteristica *visit* con el valor de *vi*.
NvoPadre (*NP*: vértice) .- Cambia el predecesor en la Arborescencia de RMC.
NvaD (*NvaD*: T_Costo) .- Modifica la Distancia del vértice.
MuestraEdo.- Despliega los valores de **Visit, Padre** y **Dist.**
AsignaStatus (*visit*: Lógico,*pre*: vértice, *NvaD*: T_Costo) .- Actualiza los valores de **Visit, Padre** y **Dist.**

Ahora bien, a todos los *Estados*, uno por vértice, los organizamos en una estructura de tamaño n, que denominaremos *RUTA*. Finalmente, las operaciones básicas para el Tipo de Datos Abstracto *RUTA* son:

CreaRuta .- Crea un objeto de tipo *RUTA*.
Status (*v*: Vértice) .- Muestra el estado actual del vértice *v*.
ModificaEdo (*v*: Vértice; *visita*: Lógico; *pa*: Vértice; *dv*: Tipo_Costo) .- Cambia los valores del estado para el vértice *v* en la RUTA.
Distancia (*v*: Vértice) .- Dado un vértice *v* indica cual es su distancia actual.

CambiaDist (*v*: Vértice; *dv*: Tipo_Costo) .- Modifica la distancia del vértice dado.
SuPadre (*v*: Vértice) .- Dado un vértice *v*, indica cuál vértice es su padre (vértice predecesor en la RMC).
CambiaPadre (*v*, *NvoPa*: Vértice) .- Dado un vértice, modifica su predecesor.
FueVisitado (*v*: Vértice) .- Indica si el vértice ya ha sido revisado.
CambiaVisitado (*v*: Vértice; *NvoV*: Lógico) .- Modifica la situación de visitado, por *NvoV*, para el vértice dado *v*.

ActivaEnQ (*v*: Vértice) .- Dado un vértice *v*, mantiene un apuntador a su etiqueta temporal en la cola de prioridades.

DesActivaEnQ (*v*: Vértice) .- Desactiva el apuntador del vértice *v* en **Q**.

ImprimeRuta .- Muestra los datos almacenados en RUTA.

ImprimeRMC .- (*vs*, *vt*: Vértice) .- Despliega la RMC entre los vértices *s* y *t*.

Algoritmo de Dijkstra

A continuación describiremos el algoritmo de DIJKSTRA aplicando las operaciones de los *TDA*'s definidos anteriormente y utilizando la notación de *POO*. Los parámetros requeridos por el algoritmo son: G, red para la cual se buscará la arborescencia de RMC; s, nodo raíz de la arborescencia; n número de vértices en la red; P, estructura donde se almacenaran los estados para cada vértice.

Dijkstra (G: Red, s: Vertice, n : Entero, P: Ruta)

```
Variables
  Q : PQ;                              { Cola de Prioridades }
  v : Vertice:                         { vertice a revisar }
  NvaD : Tipo_Costo;                   { Nueva Distancia encontrada }
Inicio
  P.IniciaRuta;                        { Da valores iniciales el vector de Estados }
  P.ModificaEdo(s, -1, 0);             { Valor inicial para s }
  Q.Create;                            { Crea Cola de Prioridades }
  Q.Inserta( s , P.Distancia( s ));    { Añade la etiqueta de s en Q }
  Mientras ¬ Q.vacio
    v ⟵ Q.BorraMin;                    { Borra de Q al elemento cuya llave sea la minima }
    Para cada arco e Adyacente a v     { Revisa los vertices Adyacentes a v }
      e ⟵ v .ObtenArco
      w ⟵ e. VerticeFinal
      NvaD ⟵ P.Distancia( v ) + e. Costo
      Si ¬ P.Visitado( w )             { Si no ha sido visitado }
        P.ModificaEdo(w, v, NvaD);     { modifica los datos de w en P e }
        Q.Inserta(w, NvaD);            { inserta w en Q }
      EnOtroCaso                       { Si ya fue visitado y }
        Si NvaD < P.Distancia( w )     { la distancia encontrada es menor }
          P.ModificaEdo(w, v, NvaD);   { modifica los datos de w en P y }
          Q.DecrementaLlave(w, NvaD);  { decrementa la llave de w en Q }
        Fin Si NvaD < ...
      Fin Si ¬ P.Visitado ...
      v .SiguienteArco
    Fin Para cada arco e ...
  Fin Mientras ¬ Q.vacio
Fin Dijkstra
```

IV Algoritmo DijkstraSort

1 Reducciones

Sean **P** y **P'** dos problemas. Supongase que un ejemplar arbitrario **E** de **P** puede ser solucionado conviertiendo **E** en un ejemplar **E'** de **P'**, resolviendo para **E'** y reduciendo la solución **S'** en la solución **S** para **E**. La Figura 2 ilustra gráficamente este proceso.

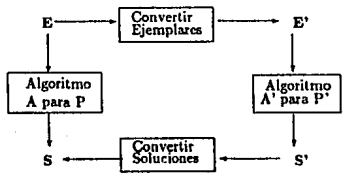


Figura 2: Reducción de **P** en **P'**

Definición

Si existen algoritmos para conviertir **E** en **E'** y **S'** en **S**, se dice entonces que existe una **reducción** de **P** en **P'** y se denota por **P** $\propto$ **P'** [16].

Generalmente, se realiza una reducción de **P** en **P'** cuando **P** es muy difícil de resolver de manera directa y **P'** es más fácil. Por ello, en cómputo, al proceso que efectúa una reducción se les denomina **Algoritmo de Reducción o Simplificación** y puede ser expresado esquemáticamente como:

Algoritmo de Simplificación (E: ejemplar)

```
Inicio
   E' ← ConvertirI (E)          { Convertir Ejemplar }
   S' ← A'( E' )                { Resolver para  P' }
   S  ← ConvertirS ( S' )       { Transformar Resultados }
   Regresa (S)
Fin
```

Ejemplo: Sean los problemas:

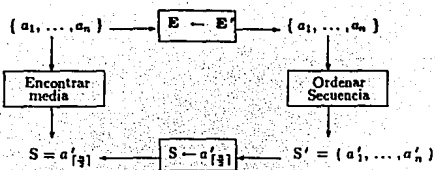


Figura 3: Reducción [Encontrar Media] $\propto$ [Ordenar Secuencia]

P: encontrar la *media* en un conjunto de n números

P': ordenar una lista de n números

El problema **P** es equivalente a buscar el $\lceil \frac{n}{2} \rceil$-ésimo elemento más pequeño del conjunto de datos. Si el conjunto está ordenado resulta muy fácil encontrar tal elemento, entonces podemos reducir $\mathbf{P} \propto \mathbf{P'}$. la Figura 3 muestra gráficamente la reducción.

Los dos algoritmos de conversión son triviales. ya que $\mathbf{E} = \mathbf{E'}$ y **S** sólo extrae el $\lceil \frac{n}{2} \rceil$-ésimo elemento de la secuencia ordenada **S'**. A continuación describimos el algoritmo de simplificación:

Algoritmo de SimplificaMedia (E: ejemplar)

Inicio

E' ⟵ E	{ Convertir Entrada }
S' ⟵ Ordenar (E')	{ Resolver para P' }
S ⟵ S' ($\lceil \frac{n}{2} \rceil$)	{ $\lceil \frac{n}{2} \rceil$-ésimo elemento de la secuencia }
Regresa (S)	

Fin

Necesitamos alguna forma de garantizar el desempeño computacional del algoritmo de simplificación para **P**, considerado que ya sabemos cual es el tiempo requerido para efectuar cada conversión y para resolver **P'**. Esta 'garantía', nos la da Kingston [16] en el Teorema 1 (TLB), el cual se ilustra en la Figura 4.

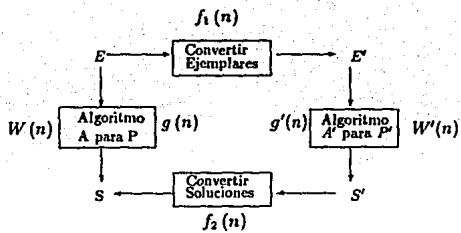


Figura 4: Teorema (Transformation Lower Bound)

Teorema 1 (Transformation Lower Bound - TLB) .

Suponga que $\mathbf{P} \propto \mathbf{P}'$ *y que los desempeños, en el peor de los casos, para los algoritmos que convierten* $\mathbf{E}$ *en* $\mathbf{E}'$ *y* $\mathbf{S}$ *en* $\mathbf{S}'$ *son* $f_1(n)$ *y* $f_2(n)$ *respectivamente, donde* n *es el tamaño de* $\mathbf{E}$. *Entonces*

a) *Por cada algoritmo con desempeño computacional* $W'(n)$ *que solucione* $\mathbf{P}'$ *existe un algoritmo para* $\mathbf{P}$ *con desempeño:*

$$W(n) = f_1(n) + W'(n) + f_2(n)$$

b) *Si* $g(n)$ *es una cota mínima (lower bound) sobre la complejidad de* $\mathbf{P}$, *en el peor de los casos, entonces una cota mínima sobre la complejidad de* $\mathbf{P}'$ *resulta ser:*

$$g'(n) = g(n) - (f_1(n) + f_2(n))$$

Nótese que el algoritmo para $\mathbf{P}$, que se menciona en el Teorema 1 (TLB), es el Algoritmo de Simplificación descrito anteriormente, ver Figura 5. Así que, el algoritmo de simplificación, para el peor de los casos, requiere un número de operaciones del orden de:

$$\begin{aligned} W(n) &= f_1(n) + W'(n) + f_2(n) + \Theta(1) \\ &= f_1(n) + W'(n) + f_2(n) \end{aligned}$$

que es lo que indica el Teorema 1 (TLB) en el inciso **(a)**.

Algoritmo de Simplificación	Desempeño Computacional
Inicio	
$\mathbf{E'}$ — ConvertirE (**E**)	$f_1(n)$
$\mathbf{S'}$ — **A'** ($\mathbf{E'}$)	$W'(n)$
S — ConvertirS ($\mathbf{S'}$)	$f_2(n)$
Regresa (**S**)	$\Theta(1)$
Fin	

Figura 5: Algoritmo Simplificacion General

El segundo aspecto relevante de las reducciones, nos la da el inciso **(b)**. Este aspecto sive para obtener cotas inferiores en cuanto a los algoritmos que resuelven un problema. Más específicamente, el inciso **(b)** nos permite decir que no es posible hallar algoritmos más rápidos a una cota previamente establecida.

En particular, si sabemos que no existen algoritmos más rápidos que una cota $g(n)$ para $\mathbf{P'}$ que $g'(n) = g(n) - (f_1(n) + f_2(n))$ ya que si existiese un algoritmo más rápido para $\mathbf{P'}$, entonces el algoritmo de transformación, sería un algoritmo para **P** más rápido que $g(n)$, lo cual es una contradicción.

Usaremos esta transformación para obtener cotas inferiores que nos indiquen que tan eficiente puede ser el desempeño computacional de los algoritmos para RMC.

2 Reducción del Problema de Ordenamiento al Problema de RMC

Sea **G** una gráfica dirigida con pesos en los arcos (red) y con un vértice distinguido v_o (nodo fuente). Considerese el siguiente problema

> $\mathbf{P'}$ Determinar para **G** *el árbol de rutas más cortas* enraizado en v_o, y encontrar una secuecia de todos los vértices de **G** ordenados según sus distancias a partir de v_o.

Al parecer, $\mathbf{P'}$ puede dividirse en dos subproblemas:

> $\mathbf{P'}_1$ Determinar la arborescencia de rutas más cortas enraizada en v_o.
>
> $\mathbf{P'}_2$ Ordenar los vértices de acuerdo a sus distancias.

En realidad el subproblema $\mathbf{P'_2}$ es resuelto por el Algoritmo de DIJKSTRA, pues éste va eliminando los vértices para los cuales la distancia desde v_o resulta ser la mínima; es decir, va sacando de la cola de prioridades las distancia de los vértices en orden no-decreciente.

Sólo se requiere hacer pequeños cambios al Algoritmo de DIJKSTRA, para obtener un algoritmo que resuelva $\mathbf{P'}$:

+ almacenar en una cola, LP, los nodos que se extraen de la cola de prioridades, **Q**;
+ regresar la cola LP una vez que **Q** sea vacío.

De esta forma tenemos un algoritmo que resuelve $\mathbf{P'}$. Estas modificaciones no alteran el desempeño computacional del algoritmo, ya que insertar un elemento en LP requiere tiempo costante. Mostrar todos los elementos de LP tarda $O(n)$. Llamemos a este algoritmo modificado: DIJKSTRAM. A continuación mostraremos que el algoritmo DIJKSTRAM es óptimo en el modelo de comparaciones.

Sea **P** (*Problema de Ordenamiento*) Ordenar los elementos de una secuencia de datos.

Denominemos por SORT a cualquier algoritmo de ordenamiento que resuelva **P**.
Sea $\ell = \{d_1, d_2, ...d_{nd}\}$ un ejemplar para **P**. Un ejemplar para $\mathbf{P'}$ debe ser una red: **G** $\simeq$ digráfica con pesos en los arcos.

Para transformar ℓ en **G** se define
v_o: vértice distinguido. (Nodo fuente);
(v_o, v_k, d_k) son los arcos de **G** y tienen costo; d_k, $\forall\, k \in [1\,..\,nd]$

Resulta claro que este proceso requiere tiempo $\Theta(n)$, en el modelo de comparaciones.

Entonces, si $\ell = \{d_1, d_2, ...d_{nd}\}$, la red es: $\mathbf{G} = (V, A, \delta)$ donde:
V es el conjunto de vertices y $|V| = nd+1 = n$
A es el conjunto de aristas y $|A| = nd$
δ es la función de costo, δ: $V \longrightarrow \mathbb{R}^+$

La Figura 6 muestra la red **G** obtenida de esta transformación. A este tipo de gráficas se les denomina *estrellas*, denotaremos por $\mathcal{E}_n$ a una gráfica estrella con n arcos y $n+1$ vértices. La Figura 7 muestra la reducción del problema de ordenamiento.

Finalmente, sea el algoritmo DIJKSTRASORT el algoritmo de simplificación para **P** que usa DIJKSTRAM:

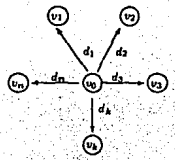


Figura 6: Red G: Gráfica Estrella $\mathcal{E}_n$

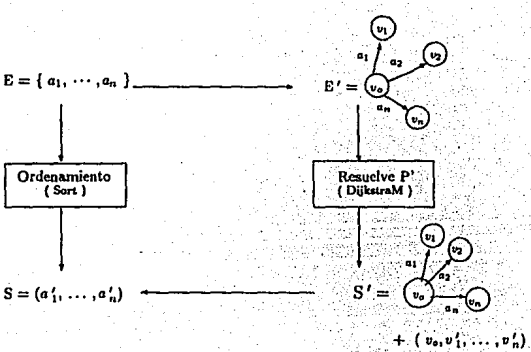


Figura 7: Reducción SORT $\propto$ DIJKSTRAM

Algoritmo DIJKSTRASORT (ℓ: SecuenciaDatos, n: integer)

Inicio

G ⟵ ConvertirE (ℓ, n)	{ Transformar de ℓ a G }
S' ⟵ DIJKSTRAM(G, v_0, n)	{ Resolver para P' }
S ⟵ ConvertirS (S')	{ Convertir Solución }
Regresa (S)	

Fin

K. Mehlhorn [23] demuestra que, tanto para el caso promedio como para el peor de los casos, $\Omega(n \log n)$ es una cota mínima para el problema de ORDENAMIENTO. Aplicando el Teorema 1 (TLB), inciso (b):

$$g'(n) = \Omega(n \cdot \log n) - (\Theta(n) + \Theta(n)) = \Omega(n \cdot \log n).$$

Entonces $\Omega(n \cdot \log n)$ representa una cota mínima para el DIJKSTRAM. Por lo tanto el algoritmo DIJKSTRAM es óptimo en el modelo de comparaciones, tanto para el caso promedio como para el peor de los casos.

El objetivo de esta tesis es mostrar que con esto no se ha dicho todo y que se pueden realizar estudios y análisis de adaptabilidad que dan más información sobre cómo implantar el algoritmo de DIJKSTRA. Es decir, pueden existir ejemplares para los cuales el algoritmo de DIJKSTRA realice menos de $\Omega(n \cdot \log n)$ operaciones elementales.

3 Análisis General del Algoritmo

En esta sección desglosaremos el algoritmo DIJSKTRAM, suponiendo que G es una gráfica estrella de tamaño n, $\mathcal{E}_n$. Enfocaremos nuestro análisis en el DIJSKTRAM ya que éste representa la parte fundamental del algoritmo DIJSKTRASORT y las conversiones tienen costo $\Theta(n)$, el cual existe en todo ejemplar de ordenamiento. La Figura 8 ilustra detalladamente la forma como se van aplicando las operaciones.

3.1 Tiempos de Ejecución

Definiciones.

- Sea t (*rutina*, q) el tiempo que tarda en efectuarse, en el peor de los casos, la operación *rutina* sobre una *Estructura* de tamaño q.

DijkstraM (G: Red, s: vértice, n : entero, P: Ruta)

Inicio
Da Valores Iniciales al Vector de Estados
Asigna valores para s { Padre = Nil, Distancia = 0, Visitado = True}
Crea Lista de Etiquetas Permanentes
Genera la Cola de Prioridades
Inserta a s en Q
Mientras $\neg$ Q.vacio { * Recorre la Cola de Prioridades }
1ª Iteracion
$v \longleftarrow Q$.BorraMin; { Se aplica con $\#Q = 1 \Rightarrow Q = \phi$ }
LP.add(v) { Se aplica con $\#LP = 0$ }
* Revisa los vertices Adyacentes a v: $v = s = v_o$ tiene nd vecinos
Desde $v_k \longleftarrow v_1$ **hasta** $v_k = v_{nd}$
$e \longleftarrow [v_o, v_k]$
$w \longleftarrow e.VerticeFinal = v_k$
$NvaD \longleftarrow P$.Distancia(v) + e.Costo
Se tiene que $\neg$ visitado(w)
P.ModificaEdo($w, v, NvaD$);
Q.Inserta($w, NvaD$); { Se aplica con $\#Q = k - 1$ }
Pasa_Siguiente_vecino
{ Al salir $\#Q = nd$ }
2ª Iteracion
$v \longleftarrow Q$.BorraMin; { Se aplica con $\#Q = nd$ }
LP.add(v) { Se aplica con $\#LP = 1$ }
{ * No hay vecinos }
⋮
k–esima Iteracion { $k > 1$ }
$v \longleftarrow Q$.BorraMin; { Se aplica con $\#Q = n - k$ }
LP.add(v) { Se aplica con $\#LP = k - 1$ }
{ * No hay vecinos }
⋮
n–esima Iteracion { $n = nd + 1$ }
$v \longleftarrow Q$.BorraMin; { Se aplica con $\#Q = 1$ }
LP.add(v) { Se aplica con $\#LP = nd$ }
{ * No hay vecinos }
Fin Mientras $\neg$ Q.vacio
Fin DijkstraM

Figura 8: Ejecución general del DIJKSTRAM

TDA	Operaciones	Tiempo de Ejecución
Cola de Prioridades **Q**		
	Genera **Q**	$\Theta(1)$
	Inserta	$t(Q.Inserta, q)$
	BorraMin	$t(Q.BorraMin, q)$
Ruta **P** *(Vector de Estados)*		
	Dar Valores Iniciales a **P**	$\Theta(n)$
	Demás operaciones	
	(Crea, ModificaEdo, Visitado ...)	$\Theta(1)$
Lista de Etiquetas Permanentes **LP**		
	Crear **LP**	$\Theta(1)$
	Insertar (add)	$\Theta(1)$

Tabla 1: Desempeño Computacional del Algoritmo DIJSKTRAM

- Denotemos por $\mathcal{T}_{\mathcal{E}}(n)$ al desempeño computacional del algoritmo DIJKSTRAM aplicado a una gráfica estrella de tamaño $n, \mathcal{E}_n$.
- Sea $\mathcal{T}_{\ell}(n)$ el desempeño computacional del algoritmo DIJKSTRASORT aplicado a una secuencia ℓ de n datos.

De acuerdo con la notación anterior, describiremos el tiempo que requiere el algoritmo DIJKSTRAM, para efectuarse. Deseamos obtener un resultado general que nos de información sobre la *implantación abstracta* del algoritmo DIJSKTRAM. Trataremos de llegar a una fórmula general que *no* dependa de una implantación concreta sino, reiteramos, de la implantación abastracta donde estamos manejando los diferentes *TDA*'s. Posteriormente, obtendremos la fórmula particular para cada implantación concreta.

La Tabla 1 nos muestra los tiempos de ejecución, para el peor de los casos, de las operaciones utilizadas por los *TDA*'s del algoritmo DIJSKTRAM. La Figura 9 muestra los tiempos efectuados al aplicar esta notación.

Sumando los tiempos anteriores, tenemos que:

$$\mathcal{T}_{\mathcal{E}}(n) = \Theta(n) + 3 \cdot \Theta(1) + t(Q.Inserta, 0) \qquad \{\text{iniciación}\}$$

$$+ \; t(Q.BorraMin, 1) + t(LP.Inserta, 0) \qquad \{\text{iteración 1}\}$$

DijkstraM (G:Red, s: vertice, n : entero, P: Ruta)

Inicio	
Da valores iniciales el vector de Estados	$\Theta(n)$
P. ModificaEdo(s)	$\Theta(1)$
Genera la Lista de Etiquetas Permanentes	$\Theta(1)$
Crea la Cola de Prioridades	$\Theta(1)$
Q. Inserta(s)	$t(Q.Inserta, 0)$
Mientras $\neg$ Q. vacio { Recorre la Cola de Prioridades }	
{ 1ª Iteracion }	
$v \longleftarrow Q$. BorraMin;	$t(Q.BorraMin, 1)$
LP. add(v)	$t(LP.add, 0)$
{ * Revisa los vertices Adyacentes a v }	
$r = s = v_o$ tiene nd vecinos	
[1.] $e \longleftarrow (v_o, v_1)$	
$w \longleftarrow e.VerticeFinal = v_1$	
$NvaD \longleftarrow P$. Distancia(v) + e. Costo	
Se tiene que $\neg$ P. visitado(w)	
P. ModificaEdo($w, v, NvaD$);	$\Theta(1)$
Q. Inserta($w, NvaD$);	$t(Q.Inserta, 0)$
Pasa_Siguiente_vecino	
[2.] $e \longleftarrow (v_o, v_2)$	
$w \longleftarrow e.VerticeFinal = v_2$	
$NvaD \longleftarrow P$. Distancia(v) + e. Costo	
Se tiene que $\neg$ visitado(w)	
P. ModificaEdo($w, v, NvaD$);	$\Theta(1)$
Q. Inserta($w, NvaD$);	$t(Q.Inserta, 1)$
Pasa_Siguiente_vecino	
⋮	
[nd] $e \longleftarrow (v_o, v_{nd})$	
$w \longleftarrow e.VerticeFinal = v_{nd}$	
$NvaD \longleftarrow P$. Distancia(v) + e. Costo	
Se tiene que $\neg$ visitado(w)	
P. ModificaEdo($w, v, NvaD$);	$\Theta(1)$
Q. Inserta($w, NvaD$);	$t(Q.Inserta, nd-1)$
{ 2ª Iteracion }	
$v \longleftarrow Q$. BorraMin;	$t(Q.BorraMin, nd)$
LP. add(v)	$t(LP.add, 1)$
{ * No hay vecinos }	$\Theta(1)$
⋮ ⋮	
{ n-esima Iteracion { $n = nd+1$ }	
$v \longleftarrow Q$. BorraMin;	$t(Q.BorraMin, 1)$
LP. add(v)	$t(LP.add, n-1)$
{ * No hay vecinos }	$\Theta(1)$
Fin Mientras $\neg$ Q.vacio	
Fin DijkstraM	

Figura 9: Iteración general del DIJSKTRAM con tiempos de ejecución

$$+ \sum_{i=1}^{nd} t(\mathit{ActualizaEstado}, n) + \sum_{i=0}^{nd-1} t(\mathit{Q.Inserta}, i) \qquad \{ \text{visita ady} \}$$

$$+ \sum_{i=1}^{nd} t(\mathit{Q.BorraMin}, i) + \sum_{i=1}^{nd} t(\mathit{LP.Inserta}, i). \qquad \{ \text{nd iteraciones} \}$$

Pero, bajo el modelo computacional de comparaciones, tenemos que:

Insertar en una estructura de tamaño 0 o 1 toma tiempo constante.

Borrar en una estructura de tamaño 1 toma tiempo constante.

$n = nd + 1 \implies nd = n - 1 \implies O(n) = O(nd)$.

$$\sum_{i=1}^{nd} t(\mathit{LP.Inserta}, i) = \Theta(n).$$

Entonces,

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(\mathit{Q.Inserta}, i) + \sum_{i=1}^{nd} t(\mathit{Q.BorraMin}, i).$$

Hemos probado el siguiente resultado:

Teorema 1 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *tiene, en el peor de los casos, desempeño computacional de,*

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(\mathit{Q.Inserta}, i) + \sum_{i=1}^{nd} t(\mathit{Q.BorraMin}, i).$$

Analicemos ahora el algoritmo DIJKSTRASORT:

Algoritmo DIJKSTRASORT (ℓ: SecuenciaDatos, n: integer)

Inicio

G' ⟵ ConvertirI (ℓ, n)	$\Theta(n)$
S' ⟵ **DijkstraM(G,** v_o **,** n **)**	$T_{\mathcal{E}}(n)$
S ⟵ ConvertirS (S')	$\Theta(n)$
Regresa (S)	$\Theta(1)$

Fin

Entonces el desempeño computacional del algoritmo DIJKSTRASORT es:

$$T_{\ell}(nd) = \Theta(n) + T_{\mathcal{E}}(n) + \Theta(n) + \Theta(1) = \Theta(n) + T_{\mathcal{E}}(n)$$

sustituyendo el valor de $\mathcal{T}_\mathcal{E}(n)$

$$\mathcal{T}_\ell(nd) = \Theta(n) + \left(\Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)\right)$$

$$= \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i) = \mathcal{T}_\mathcal{E}(n)$$

De donde obtenemos el siguiente resultado:

Teorema 2 *El algoritmo* DIJKSTRASORT, *aplicado a una secuencia de n datos, tiene desempeño computacional en el peor de los casos de:*

$$\mathcal{T}_\ell(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

Finalmente, observamos que:

Corolario 3 *Los algoritmos* DIJSKTRASORT *y* DIJKSTRAM, *en redes estrella, tienen el mismo desempeño computacional en el peor de los casos.*

V Adaptabilidad en el Problema de la Ruta Más Corta

1 Análisis e Ilustración Experimental

Para cada una de las implantaciones concretas, del mismo TDA *Cola de Prioridades*, analizaremos el comportamiento de los algoritmos DIJKSTRAM y DIJKSTRASORT, descritos en el capítulo anterior. En el análisis de algoritmos, trabajaremos con la implantación abstracta descrita anteriormente, la especificación de los TDA para las etiquetas temporales se hará en diferentes estructuras de datos: *Listas Ligadas, Listas Doblemente Ligadas con apuntador, Colas Binomiales y Fibonacci-Heaps.*

Con el objetivo de ilustrar nuestros teoremas obtenidos mediante el análisis matemático de algoritmos, implantaremos la reducción del problema de ORDENAMIENTO utilizando el algoritmo DIJKSTRAM, denominada DIJKSTRASORT y analizaremos el comportamiento de los programas.

Efectuaremos un análisis sobre los *Heaps Binomiales y Colas Binomiales*, pero no la implantación. Todas las demás fueron programadas como implantaciones concretas del mismo TDA Cola de Prioridades.

Realizaremos *experimentos* en los cuales ejemplares de secuencias ℓ para DIJKSTRASORT y gráficas $\mathcal{E}_n$ para DIJKSTRAM.

En particular, para ilustrar el comportamiento adaptivo (o su inexistencia) realizaremos el análisis de algoritmos o corridas empíricas en cuatro tipos de secuencias ℓ:

- **I.** Secuencias ℓ donde los elementos están totalmente ordenados.
- **II.** Secuencias ℓ donde los elementos están ordenados descendentemente.
- **III.** Secuencias ℓ con distintos niveles de desorden.
- **IV.** Secuencias donde ℓ es una lista obtenida aleatoriamente bajo una distribución uniforme.

En lo que sigue, ℓ es una secuencia de elementos pertenecientes a un orden total, $|\ell|$ denota el número de elementos en ℓ y si $|\ell| = nd$, entonces $|V|$ (el número total de vértices en la red) es $(nd+1)$ y por sencillez denotaremos $|V| = n$.

En cada sección compararemos los resultados obtenidos en los análisis teóricos con varias ejecuciones de las implantaciones concretas programadas en el Lenguaje Orientado a Objetos EIFFEL [24, 25, 26, 28, 27] y ejecutadas sobre una red de estaciones de trabajo SUN SPARC STATION [40, 39, 41, 38].

Cabe mencionar que la descripción detallada de los tipos de datos concretos manipulados en este capítulo se encuentra en el Apendice **B**.

2 Monitoreo

Para medir el consumo de recursos de cómputo (en particular, el tiempo de ejecución) de nuestros programas, realizamos un monitoreo para revisar el comportamiento del algoritmo DIJKSTRASORT y contar el número de operaciones elementales que se efectúan en el programa tanto en **Q**, la cola de prioridades; como en **LP**, la cola de vértices y **G**, la red. Definimos los siguientes contadores de operaciones elementales:

Contador	para
Koe_PQ	la Cola de Prioridades, **Q**;
Koe_LP	la cola de vertices, **LP**;
Koe_G	la Red, **G**;
KOE	el algoritmo, **KOE = Koe_PQ + Koe_LP + Koe_G**

Para ilustrar el comportamiento de la ejecución del algoritmo, en distintos ejemplares con diferentes niveles de dificultad desde el punto de vista de ordenamiento, elegimos una familia de secuencia de datos, la cual denominamos secuencia de datos organizada en forma *zig-zag* que se define de la siguiente manera:

Definición

Sea $\ell = \{ x_1, x_2, \ldots, x_n \}$ una secuencia ordenada de n datos.

La secuencia de datos organizada en zig-zag para ℓ es:

$$\ell' = \{x_{\lceil \frac{n}{2} \rceil}, \ldots, x_2, x_{n-1}, x_1, x_n\}$$

Ejemplo 1

Sea $\ell = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$, entonces la secuencia de datos organizada en zig-zag para ℓ es: $\ell' = \{ 5, 4, 6, 3, 7, 2, 8, 1, 9 \}$

Ahora bien, ¿por qué no pensar en hacer un $zig - zag$ más global?; esto es, en lugar de ir intercalando un elemento, ¿por qué no intercalamos un bloque de elementos, ordenados, de tamaño d? Esto nos lleva a la siguiente definición.

Definición.

Sea $\ell = \{ b_1, b_2, \ldots, b_k, B, \beta_1, \beta_2, \ldots, \beta_k \}$ una secuencia de bloques ordenados donde el tamaño de cada bloque b_i, β_i es d y el tamaño del bloque B es, $|B| = (n \bmod d)$, con $n = |\ell|$. Diremos que ℓ' está **organizada en la forma zig-zag-d** si: $\ell' = \{B, b_k, \beta_k, \ldots, b_2, \beta_2, b_1, \beta_1, \}$

Otra forma de definir este tipo de secuencias *zig-zag-d* es:

Definición.

Sea $\ell = \{ x_1, x_2, \ldots, x_n \}$ una secuencia de datos ordenados de tamaño n. La **secuencia de datos organizada en forma zig-zag-d** para ℓ es:

$\ell' = \{\ldots, x_{d+1}, \ldots, x_{2d+1}, x_{n-2d+1}, \ldots, x_{n-d}, x_1, x_2, \ldots, x_d, x_{n-d+1}, \ldots, x_{n-1}, x_n\}$

Ejemplo 2

Sea $\ell = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \}$ y $d = 2$ entonces la secuencia de datos organizada en **zig-zag-2** para ℓ es: $\ell' = \{ 5, 6, 3, 4, 7, 8, 1, 2, 9, 10 \}$

Ejemplo 3

Sea $\ell = \{ x \in \mathbf{N} / 1 \leq x \leq 20 \}$ y $d = 3$ entonces la secuencia de datos organizada en **zig-zag-3** para ℓ es:

$\ell' = \{ 10, 11, 7, 8, 9, 12, 13, 14, 4, 5, 6, 15, 16, 17, 1, 2, 3, 18, 19, 20 \}$.

La Figura 10 muestra gráficamente la distribución de los datos.

Ejemplo 4

Sea $\ell = \{ x \in \mathbf{N} / 1 \leq x \leq 20 \}$ y $d = 1$ entonces la secuencia de datos organizada en **zig-zag-1** para ℓ es:

$\ell' = \{ 10, 11, 9, 12, 8, 13, 7, 14, 6, 15, 5, 16, 4, 17, 3, 18, 2, 19, 1, 20 \}$.

La Figura 11 muestra la distribución de los datos.

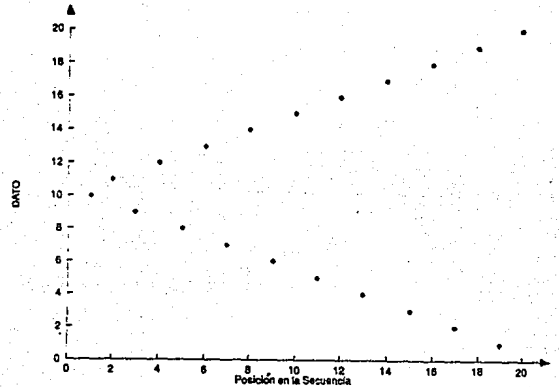


Figura 10: *Zig-zag-3* para $n = 20$.

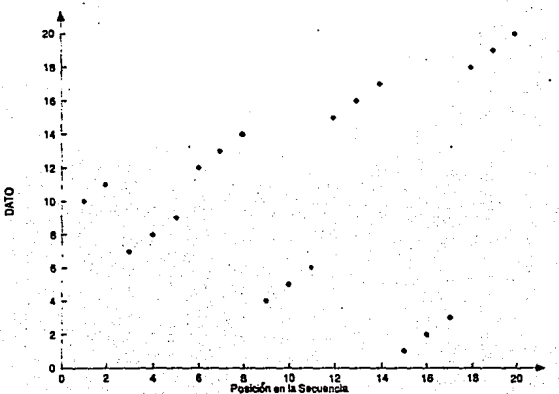


Figura 11: *Zig-zag-1* para $n = 20$.

Esta familia de secuencias tiene caracteristicas muy especiales:

1) Si $d = 1$ tenemos una secuencia organizada en forma $zig - zag$.

2) Cuando $d = 1$ la secuencia ℓ' está *muy* desordenada, de hecho, resulta ser uno de los peores casos para los tipos de datos concretos que hemos implantado en EIFFEL [24, 25, 26, 28, 27].

3) Cuando $d = n$ la secuencia ℓ' queda ordenada, esto es: $\ell = \ell'$

Así que, tenemos una familia de ejemplares fácil de construir con la que podemos revisar el desempeño computacional de nuestros programas para el algoritmo DIJKSTRASORT y diferentes *niveles* de desorden en la secuencia de datos de entrada.

3 Listas Simplemente Ligadas

En esta sección desarrollaremos el análisis para DIJKSTRASORT y DIJKSTRAM, utilizando, para las etiquetas temporales, la versión más simple de Colas de Prioridades con Listas Ligadas. A esta implantación concreta la denominamos, en el Apendice **B**, *clase pq_lsl*. Iniciamos dando una breve explicación de las operaciones en una cola con i elementos.

Descripción de las operaciones en una lista con i elementos, para la *clase pq_lsl*.

MinH .- Apuntador al elemento con llave mínima.

Inserta .- Añade un nuevo elemento al final de la Lista, si el nuevo elemento es menor que el mínimo, modifica **MinH**. Requiere tiempo $\Theta(1)$.

EncuentraMin. Pregunta por el apuntador *MinH*, requiere tiempo $\Theta(1)$.

BorraMin .- Almacena en una variable temporal al elemento mínimo, lo elimina de la lista, después busca en *toda* la cola al nuevo elemento con mínima etiqueta y regresa el elemento que fuera el mínimo. Requiere tiempo: $O(i)$.

Decrementa_LLave .- Recorre toda la lista hasta encontrar el vértice a cambiar, le decrementa la llave y verifica si el elemento mínimo se modifica. Requiere tiempo: $O(i)$.

Elimina .- Borra un elemento de la cola, supone conocida la posición de tal elemento.

3.1 Tiempos de Ejecución

Aplicando el Teorema 1 y evaluando el tiempo que gastan las operaciones, tenemos:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} T(Q.Inserta, i) + \sum_{i=1}^{nd} T(Q.BorraMin, i)$$

$$\Longrightarrow T_{\mathcal{E}}(n) = \Theta(n) + O(nd-1) + O\left(\sum_{i=1}^{nd} i\right)$$

$$\Longrightarrow T_{\mathcal{E}}(n) = \Theta(n) + \Theta(nd) + O\left(\tfrac{nd(nd+1)}{2}\right)$$

$$\Longrightarrow T_{\mathcal{E}}(n) = \Theta(n) + \Theta(nd) + O(nd^2)$$

$$\Longrightarrow T_{\mathcal{E}}(n) = \Theta(n) + \Theta(n) + O(n^2) = O(n^2).$$

Entonces, obtenemos los siguientes resultados.

Teorema 4 *El algoritmo* DIJKSTRAM *implantando con la clase pq_lsl y aplicado a una gráfica estrella* $\mathcal{E}_n$, *requiere tiempo* $O(n^2)$ *en el peor de los casos.*

Corolario 5 *El algoritmo* DIJKSTRASORT *implantando con la clase pq_lsl y aplicado a una secuencia de datos* ℓ_n, *requiere tiempo* $O(n^2)$ *en el peor de los casos.*

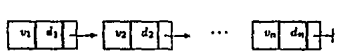
3.2 Análisis de Adaptabilidad

Observamos que:

1) El desempeño computacional del DIJKSTRAM, utilizando este TDC, depende únicamente de $t(Q.BorraMin, i)$

2) Al almacenar ℓ en la cola **Q** los datos quedan organizados de la misma manera en que se encontraban en ℓ.

Ejemplo: Si $\ell = \{ d_1, d_2, \ldots, d_n \}$ entonces,



Caso I. Si ℓ está ordenada en forma ascendente $\Rightarrow$ **Q** está ordenada.

Cuando la operación **BorraMin** se ejecuta, el nuevo elemento mínimo queda en la primera posición de la cola, pero el algoritmo lo busca en todo **Q**. Entonces, si la lista tiene i datos, la operación **BorraMin** es $\Omega(i)$. Por otra parte, sabemos que **Q** tiene al menos nd elementos ($nd = O(n)$) además esta operación se ejecuta n veces, y como i va de 1 a nd, finalmente tenemos que el tiempo de ejecución de este algoritmo es $\Theta(n^2)$.

Es claro que el algoritmo DIJKSTRASORT *no es adaptivo* pues en este caso, por ejemplo, no *aprovecha* que la lista ya está ordenada.

Caso II. Si ℓ está ordenada en forma descendente $\Rightarrow$ **Q** también está ordenada.

Una vez que la operación **BorraMin** se efectúa, el nuevo mínimo queda en la última posición de la lista, posición i. El algoritmo lo busca desde el inicio de **Q**. De manera similar al caso anterior, se tiene que el desempeño computacional del algoritmo es $\Theta(n^2)$.

Caso III. Si ℓ ha sido obtenida aletoriamente bajo una distribución uniforme.

El comportamiento esperado del algoritmo es:

$$\mathbf{E}[\mathcal{T}_\ell(n)] = \mathbf{E}\left[\Theta(n) + \sum_{i=1}^{nd-1} \mathcal{T}(Q.Inserta, i) + \sum_{i=1}^{nd} \mathcal{T}(Q.BorraMin, i)\right]$$

Sabemos que: $\mathbf{E}[\sum(\cdot)] = \sum(\mathbf{E}[\cdot])$, por lo que

$$\mathbf{E}[\mathcal{T}_\ell(n)] = \mathbf{E}[\Theta(n)] + \mathbf{E}\left[\sum_{i=1}^{nd-1} \mathcal{T}(Q.Inserta, i)\right] + \mathbf{E}\left[\sum_{i=1}^{nd} \mathcal{T}(Q.BorraMin, i)\right]$$

$$= \mathbf{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathbf{E}[\mathcal{T}(Q.Inserta, i)] + \sum_{i=1}^{nd} \mathbf{E}[\mathcal{T}(Q.BorraMin, i)]$$

$$= \mathbf{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathbf{E}[\Theta(1)] + \sum_{i=1}^{nd} \mathbf{E}[\Theta(i)]$$

$$\Longrightarrow \mathbf{E}[\mathcal{T}_\ell(n)] = \Theta(n) + \Theta(n) + \Theta(n^2) = \Theta(n^2).$$

Por tanto, el desempeño computacional del algoritmo en el caso promedio es $\Theta(n^2)$ operaciones elementales.

Operaciones		
Número	Nombre	Costo
n+1	**Inserta**	$\Theta(1)$
n+1	**BorraMin**	$\Theta(\#Q)$
n+1	**Vacio?**	$\Theta(1)$
1	**Principio**	$\Theta(1)$
1	**Crea_PQ**	$\Theta(1)$

Tabla 2: Número de Operaciones para **Q** usando *pq_lsl*.

Podemos concluir los siguientes resultados:

Teorema 6 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *e implantado con la clase pq_lsl, requiere tiempo* $\Theta(n^2)$.

Corolario 7 *El algoritmo* DIJKSTRASORT *implantado con la clase pq_lsl y aplicado a una secuencia* ℓ *de* n *elementos, requiere tiempo* $\Theta(n^2)$.

Afirmación 8 *El algoritmo* DIJKSTRASORT *implantado con la clase pq_lsl no es adaptivo.*

3.3 Resultados Empíricos

Antes de pasar a los resultados obtenidos al ejecutar una implatación concreta del algoritmo DIJKSTRASORT, implantado con la clase *pq_lsl*, observemos que podemos refinar los resultados anteriores en términos del análisis del número de operaciones elementales, no sólo en el modelo teórico de comparaciones. Revisaremos, nuevamente, el comportamiento general del algoritmo DIJKSTRAM, aplicado a una gráfica estrella $\mathcal{E}_n$.

Para **Q**, la cola de prioridades, el algoritmo requiere un número de operaciones como se indica en la Tabla 2, donde **#Q** representa el número de elementos en **Q**.

La operación **Inserta**, Figura 12, realiza dos operaciones elementales y si la llave del nuevo elemento es menor que la del mínimo, se actualiza el **MinH**, lo cual requiere otra operación elemental. En el mejor de los casos, el mínimo no se actualiza, entonces **Inserta** requiere dos operaciones elementales. En el peor de los casos, el mínimo se modifica en cada inserción, entonces **Inserta** requiere de tres operaciones elementales.

```
Inserta(x: dato)
 InicioProceso
   Add(x);                          { agrega dato en la ultima posicion }
   Si  Llave(x) < EncuentraMin
   Entonces  CambiaMin;             { Actualizamos el minimo }
   Fin-Si
 FinProceso
```

Figura 12: Proceso Inserta.

```
BorraMin: Vertice
 variables
   VerticeMin : Vertice;
 InicioProceso
   VerticeMin := VerticeMinimo;     { lo guardamos para no perderlo }
   Elimina;                         { Quita el nodo que contiene al minimo de H }
   BuscaNvoMin;                     { Actualizamos el minimo }
   Result := VerticeMin;            { y regresamos el minimo anterior }
 FinProceso
```

Figura 13: Proceso BorraMin

En realidad, la operación **BorraMin** (Figura 13) se realiza en tiempo $\Theta(i)$, donde i es el tamaño de la lista. Las operaciones **VerticeMinimo** y **Elimina** gastan tiempo constante, al igual que la operación que regresa el valor de la función. Cada **BuscaNvoMin** efectúa un **Principio** y un **CambiaMin**, ambos requieren tiempo constante, pero **BuscaNvoMin** recorre toda la cola para garantizar que ha encontrado el **VerticeMinimo**, por lo tanto, cada **BorraMin** realiza $(i+4)$ operaciones elementales.

Contemos ahora, el total de operaciones elementales que realiza el proceso **BorraMin** en el algoritmo DIKJSTRAM. Al empezar el ciclo principal del algoritmo, **G** tiene 1 dato, para la segunda iteración **G** tiene n datos, después va disminuyendo en uno el número de datos en la cola. Finalmente, tenemos que el número total de operaciones elementales se ejecutan en el proceso **BorraMin** son, $\mathbf{Koe\_BMin} = 1 + \frac{n\cdot(n+1)}{2} + 4\cdot(n+1)$.

Entonces tenemos que, en el mejor de los casos:

$$\mathbf{Koe\_PQ} = (n+1)\cdot 2 + (n+1)\cdot 1 + 2 + \left(1 + \frac{n\cdot(n+1)}{2} + 4\cdot(n+1)\right)$$

$$= 7\cdot(n+1) + \frac{n\cdot(n+1)}{2} + 3 = \frac{n\cdot(n+1)}{2} + 7\cdot n + 10.$$

Para el peor de los casos:

$$\mathbf{Koe\_PQ} = (n+1)\cdot 3 + (n+1)\cdot 1 + 2 + \left(1 + \frac{n\cdot(n+1)}{2} + 4\cdot(n+1)\right)$$

$$= 7\cdot(n+1) + \frac{n\cdot(n+1)}{2} + 3 = \frac{n\cdot(n+1)}{2} + 7\cdot n + 10 + (n+1).$$

Podemos decir que:

$$\mathbf{Koe\_PQ} = \frac{n\cdot(n+1)}{2} + 7\cdot n + K \quad \text{donde: } 10 \leq K \leq (n+1).$$

Para **LP**, la cola de vértices, el algoritmo realiza $(n+1)$ **Inserta** y un **Crea_LP**. Cada una gasta un número constante de operaciones elementales, así que: $\mathbf{Keo\_LP} = n + 2$. Para la Red **G**, el algoritmo sólo lee los costos de las aristas, así que $\mathbf{Keo\_G} = n$.

Finalmente,

$$\mathbf{KOE} = \mathbf{Koe\_PQ} + \mathbf{Koe\_PQ} + \mathbf{Keo\_LP} + \mathbf{Keo\_G}$$

$$= \left[\frac{n\cdot(n+1)}{2} + 7\cdot n + K\right] + (n+2) + (n) = \frac{n\cdot(n+1)}{2} + 9\cdot n + C$$

donde: $12 \leq C \leq (n+1)$.

Los resultados obtenidos de la ejecución del programa nos indican que el algoritmo DIJKSTRAM efectua un número de operaciones elementales dado por:

$$\mathbf{KOE} = \frac{n\cdot(n+1)}{2} + 9\cdot n + K, \quad K \geq 10.$$

Así que, tenemos que el número de operaciones elementales, **KOE**, es del orden de $\Theta(n^2)$, lo cual concuerda con el Teorema 5.

La gráfica de la Figura 14 muestra el crecimiento de número de operaciones elementales del algoritmo DIJKSTRASORT programado con listas simplemente ligadas, como TDC para la cola de prioridades, con respecto al tamaño de las secuencias a ordenar.

Para la gráfica de la Figura 14, se tiene que el eje X representa el tamaño de la secuencia a ordenar, el eje Y representa el número de operaciones elementales que requiere el DIJKSTRASORT para ordenar la secuencia, además:

+ koe(x) = **KOE** = $\frac{x\cdot(x+1)}{2} + 9\cdot x + 12$.
+ "koe.dj" son los **KOE**'s obtenidos al ejecutar el DIJKSTRASORT en una secuencia de datos organizada en forma zig-zag- j, $j = 1, 2, ..., n$.

La gráfica de la Figura 15 muestra el crecimiento del número de operaciones elementales, para la cola de prioridades, **Q** del algoritmo DIJKSTRASORT con respecto al tamaño de las secuencias a ordenar. Estos resultados fueron obtenidos en nuestros programas al implantar el DIJKSTRASORT con la clase *pq_lsl*.

Para la gráfica de la Figura 15, el eje X representa el tamaño de la secuencia a ordenar, el eje Y es el número de operaciones elementales que utiliza **Q** al ejecutar el DIJKSTRASORT, además:

+ koe_pq(x) = **KOE_PQ** = $\frac{x\cdot(x+1)}{2} + 7\cdot x + 10$.
+ "ko_pq.dj" son los **KOE_PQ**'s obtenidos al ejecutar el DIJKSTRASORT en una secuencia de datos organizada en forma zig-zag- j, $j = 1, 2, ..., n$.

Se ilustra con estás gráficas, Figuras 14 y 15, el crecimiento cuadrático del número de operaciones elementales, con respecto al número de datos en la secuencia a ordenar, independientemente de la forma de los datos, como lo establece el Teorema 5.

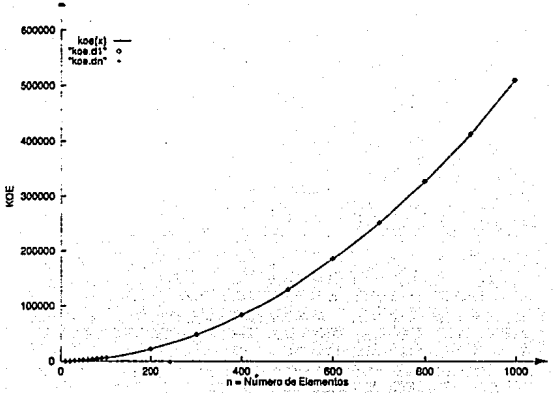


Figura 14: KOE para el DIJKSTRASORT utilizando *pq_lsl*.

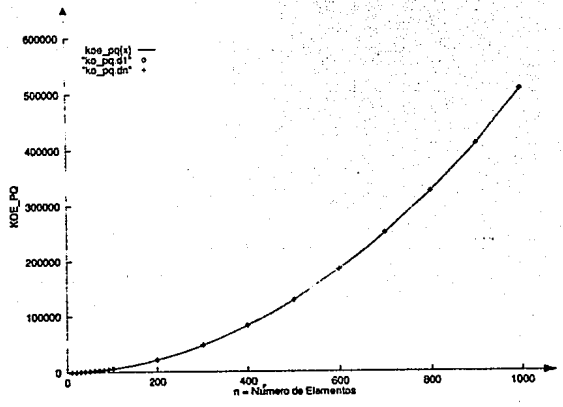


Figura 15: KOE_PQ para el DIJKSTRASORT utilizando *pq_lsl*.

4 Listas Doblemente Ligadas con apuntador

En esta sección desarrollaremos el análisis para los algoritmos DIJKSTRAM y DIJKSTRASORT, utilizando, para las etiquetas temporales, la estructura de datos *Listas Doblemente Ligadas con Apuntador* e implantando una operación de inserción local. En el Apendice B, a esta implantación concreta la denominamos *clase pq_ldl*. Empezamos dando una breve explicación de las operaciones en una cola con i elementos.

Descripción de las operaciones en una lista con i elementos , para la *Clase pq_ldl*:

ui .- Apuntador al último elemento insertado.

Inserta(x : dato) .- Busca en **Q** el lugar que le corresponde al dato de acuerdo a su llave, a partir de **ui**. **Inserta** depende del número de elementos que hay entre ui y la posición correcta de x , sea d_x tal distancia. Por tanto, **Inserta** gasta tiempo $O(d_x)$. Nótese que en el mejor de los casos $d_x = 0$ y en el peor de los casos $d_x = i$.

BorraMin .- El mínimo elemento siempre está al principio de la cola, la operación requiere tiempo constante: $\Theta(1)$.

DecrementaLLave .- Debe buscar el vértice para el cual se hará el cambio, decrementar la llave y verificar si **Q** queda en orden, si no es así debe reubicar el dato que ha quedado en desorden. Requiere tiempo $\Theta(i)$.

EncuentraMinimo .- Como el primer elemento de la cola será siempre el elemento mínimo, esta operación se ejecuta en tiempo constante, $\Theta(1)$.

4.1 Tiempos de Ejecución

Al aplicar el Teorema 1, tenemos que, utilizando esta clase, el desempeño computacional del Algoritmo DIJKSTRAM es:

$$T_E(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

$$\Longrightarrow T_E(n) = \Theta(n) + \sum_{i=0}^{nd-1} t(Q.Inserta, i) + \Theta(nd)$$

$$\Longrightarrow T_E(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \Theta(n)$$

$$\Longrightarrow T_E(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i).$$

Corolario 9 *El Algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *e implantando con la Clase* pq_ldl *tiene desempeño computacional, en el peor de los casos, de:*

$$T_E(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i).$$

4.2 Análisis de Adaptabilidad

Caso I Si ℓ está ordenada en forma ascendente $\Longrightarrow d_x = 0 \ \forall \ x \in \ell$.

Cada nuevo x a insertar se añade exactamente a la derecha de ui, entoces:

$$t(Q.Inserta, i) = \Theta(1) \quad \forall \ x \in \ell.$$

Por tanto: $T_\ell(n) = \Theta(n) + \sum_{i=0}^{nd-1} t(Q.Inserta, i)$

$$\Longrightarrow T_\ell(n) = \Theta(n) + \Theta(nd) \quad \Longrightarrow T_\ell(n) = \Theta(n).$$

Corolario 10 *El Algoritmo* DIJKSTRASORT *aplicado a una secuencia* ℓ *de* n *datos, ordenados en forma ascendente, e implantado con el TDC pq_ldl requiere tiempo* $\Theta(n)$.

Caso II Si ℓ está ordenada en forma descendente $\Longrightarrow d_x = 0 \ \forall \ x \in \ell$.

Cada nuevo x a insertar se añade exactamente a la izquierda de ui, entonces:

$$t(Q.Inserta, i) = \Theta(1) \quad \forall \ x \in \ell.$$

Por tanto: $T_\ell(n) = \Theta(n) + \sum_{i=0}^{nd-1} t(Q.Inserta, i)$

$$\Longrightarrow T_\ell(n) = \Theta(n) + \Theta(nd) \quad \Longrightarrow T_\ell(n) = \Theta(n).$$

Corolario 11 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de* n *datos en orden descendente e implantado con el TDC* pq_ldl *requiere tiempo* $\Theta(n)$.

Caso III Si ℓ ha sido obtenida aletoriamente bajo una distribución uniforme.

El comportamiento esperado del algortimo es:

$$\mathrm{E}[\mathcal{T}_t(n)] = \mathrm{E}\left[\Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta,i) + \sum_{i=1}^{nd} t(Q.BorraMin,i)\right]$$

$$= \mathrm{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathrm{E}[t(Q.Inserta,i)] + \sum_{i=1}^{nd} \mathrm{E}[t(Q.BorraMin,i)]$$

$$= \Theta(n) + \sum_{i=1}^{nd-1} \Theta(i) + \sum_{i=1}^{nd} \Theta(1) = \Theta(n) + \Theta(n^2) + \Theta(n)$$

$$\Longrightarrow \mathrm{E}[\mathcal{T}_t(n)] = \Theta(n^2).$$

Porque $\mathrm{E}[t(Q.Inserta,i)] = \Theta(i)$ ya que:

$$\mathrm{E}[t(Q.Inserta,i)] = \sum_{j=1}^{i}\sum_{k=0}^{i}\left(\mathbf{Prob}\left[\begin{array}{c}\text{el apuntador}\\ \text{ul está en la}\\ \text{posición } i\end{array}\right]\mathbf{Prob}\left[\begin{array}{c}\text{el nuevo elemento}\\ \text{debe insertarse en}\\ \text{la posición } k\end{array}\right]\right)\cdot|j-k|$$

$$= \sum_{j=1}^{i}\sum_{k=0}^{i}\left(\frac{1}{i}\right)\left(\frac{1}{i+1}\right)\cdot|j-k| = \Theta(i).$$

Corolario 12 *El desempeño computacional del algoritmo* DIJKSTRASORT *aplicado a una secuencia de n datos y obtenida aleatoriamente de una distribución uniforme e implantado con el TDC pq_ldl es:* $\Theta(n^2)$.

Teorema 13 (Desempeño Computacional del DIJSKTRASORT**)** *El algoritmo* DIJSKTRASORT *aplicado a una secuencia de nd datos, ℓ_n e implantado con la clase pq_ldl tiene desempeño computacional, en el peor de los casos de:*

$$\mathcal{T}_t(n) = \Theta(n) + 2\cdot\min\{Inv^-(\ell_n), Inv^+(\ell_n)\}$$

donde: $\ell_n = \{x_1, x_2, \ldots, x_n\}$

$Inv^+(\ell_n)$ *representa el número de inversiones* [5] *en orden creciente, esto es:*

$$Inv^+(\ell_n) = |\{(i,j) / 1 \leq i < j \leq n \ \ y \ \ x_i > x_j\}|.$$

[5] *Inv* es una medida del desorden, la cual se describe con más detalle en el apéndice E.

$Inv^{-}(\ell_n)$ representa el número de inversiones en orden decreciente:

$$Inv^{-}(\ell_n) = |\{(i,j) / 1 \geq i > j \geq n \ y \ x_i < x_j\}|.$$

Demostración.

Supongamos que $\ell_n = \{x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n\}$ y que $\mathcal{E}_n$ es la gráfica estrella que resulta de transformar el ejemplar ℓ_n al aplicar el algoritmo DIJKSTRASORT.

Tenemos que el desempeño computacional del DIJSKTRAM, en el peor de los casos, es:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

y como $t(Q.BorraMin, i) = 1 \implies \sum_{i=1}^{nd} t(Q.BorraMin, i) = \Theta(n)$,

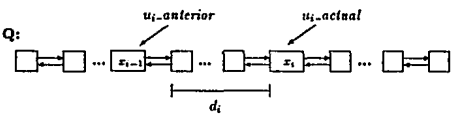
entonces: $T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i)$.

Pero **Insertar** el i-ésimo elemento, x_i, en **Q** depende de d_i, que es la distancia entre el último elemento insertado y la posición correcta de x_i, y aquí tenemos dos casos: cuando x_i es mayor que el último elemento insertado o cuando es menor. Analizaremos ambos.

Caso 1

Si x_i es mayor que el último elemento insertado. Esto es: $x_i > x_{i-1}$.

La siguiente figura nos da una representación gráfica de este caso.



Sea $X_I(i)$ el conjunto de todos los elementos que llegaron antes que x_i y que son mayores que x_i. Esto es: $X_I(i) = \{x_j / 1 \leq j \leq i-1 \ y \ x_j > x_i\}$. La cardinalidad de $X_I(i)$ es una medida del desorden denominada $Inv(x_i)$ [7, 8, 35].

Sea $X_A(i)$ el conjunto de todos los elementos que llegaron antes que x_i, que son menores que x_i y mayores que x_{i-1}. Esto es: $X_A(i) = \{x_j / 1 \leq j < i-1 \ y \ x_{i-1} < x_j < x_i\}$.

Es claro que $d_i \leq |X_A(i)| \leq |X_I(i)| = Inv(x_i) \implies d_i \leq Inv(x_{i-1})$.

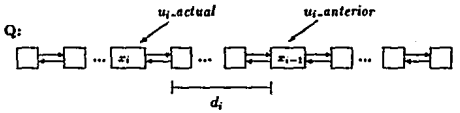
Además, sabemos que, $Inv(\ell_n) = \sum_{i=1}^{n} Inv(x_i)$,

por lo tanto: $d_i \leq Inv(x_{i-1}) \implies \sum_{i=1}^{n} d_i \leq Inv(\ell_n)$.

Caso 2

Si x_i es menor que el último elemento insertado. Esto es: $x_i < x_{i-1}$.

La siguiente figura nos muestra una representación gráfica de este caso.



Sea $X_B(i)$ el conjunto de todos los elementos que llegaron antes que x_i, que son mayores que x_i y menores que x_{i-1}. Es decir: $X_B(i) = \{x_j / 1 \leq j < i-1 \text{ y } x_{i-1} > x_j > x_i\}$.

Es claro que: $d_i \leq \|X_A(i)\| \leq \|X_B(i)\| = Inv(x_i) \implies d_i \leq Inv(x_i)$.

Sabemos que, $Inv(\ell_n) = \sum_{i=1}^{n} Inv(x_i)$.

Por lo tanto: $d_i \leq Inv(x_i) \implies \sum_{i=1}^{n} d_i \leq Inv(\ell_n)$.

Como d_i y las inversiones son números positivos: $d_i \leq Inv(x_i) + Inv(x_{i-1})$

entonces, $\sum_{i=1}^{n} d_i \leq \sum_{i=1}^{n} [\, Inv(x_i) + Inv(x_{i-1}) \,]$

$$\implies \sum_{i=1}^{n} d_i \leq 2 \cdot Inv(\ell_n).$$

Regresando al análisis de la operación **Inserta**, y denotando inversiones en el orden ascendente como $Inv^+(\ell)$, tenemos que:

$$\sum_{i=1}^{n} t(Q.Inserta, i) \leq \sum_{i=1}^{n} d_i \leq \sum_{i=1}^{n} 2 \cdot Inv^+(\ell).$$

Redefiniendo inversiones en el orden descendente como $Inv^{-}(\ell)$, similarmente, se obtiene que $\sum_{i=1}^{n} d_i \leq 2 \cdot Inv^{-}(\ell_n)$ y que $\sum_{i=1}^{n} t(Q.Inserta, i) \leq \sum_{i=1}^{n} 2 \cdot Inv^{-}(\ell_n)$.

Finalmente, $\mathcal{T}_{\mathcal{E}}(n) = \Theta(n) + 2 \cdot \min\{ Inv^{-}(\ell_n), Inv^{+}(\ell_n) \}$.

Como ejemplo de que el Teorema 13 es, en realidad, informativo y aplicable, re-verifiquemos que los resultados 10, 11, 12 son derivables de él.

Para el Corolario 10:

Aplicando el Teorema anterior: $\mathcal{T}_{\ell}(n) = \Theta(n) + 2 \cdot \min\{ Inv^{-}(\ell_n), Inv^{+}(\ell_n) \}$.

Las $Inv^{+}(\ell_n) = 0$ y $Inv^{-}(\ell_n) = O(n^2)$, entonces:

$$\mathcal{T}_{\ell}(n) = \Theta(n) + 2 \cdot 0 = \Theta(n) \implies \mathcal{T}_{\ell}(n) = \Theta(n).$$

Para el Corolario 11:

Aplicando el Teorema 13 : $\mathcal{T}_{\ell}(n) = \Theta(n) + 2 \cdot \min\{ Inv^{-}(\ell_n), Inv^{+}(\ell_n) \}$.

Las $Inv^{+}(\ell_n) = O(n^2)$ y $Inv^{-}(\ell_n) = 0$, entonces:

$$\mathcal{T}_{\ell}(n) = \Theta(n) + 2 \cdot 0 = \Theta(n) \implies \mathcal{T}_{\ell}(n) = \Theta(n).$$

Para el Corolario 12:

Aplicando el teorema del desempeño computacional para el DIJKSTRASORT:

$$\mathcal{T}_{\ell}(n) = \Theta(n) + 2 \cdot \min\{ Inv^{-}(\ell_n), Inv^{+}(\ell_n) \}.$$

Como $\mathbf{E}[Inv^{+}(\ell_n)] = \Theta(n^2/4)$ y $\mathbf{E}[Inv^{-}(\ell_n)] = \Theta(n^2/4)$,
entonces: $\mathbf{E}[\mathcal{T}_{\ell}(n)] = \Theta(n) + 2 \cdot \Theta(n^2/4)$
$\implies \mathbf{E}[\mathcal{T}_{\ell}(n)] = \Theta(n^2)$.

Teorema 14 *El algoritmo* DIJKSTRASORT *implantado con el TDC pq_ldl es adaptivo con respecto a la medida del desorden denominada Inversiones.*

4.3 Resultados Empíricos

Nuevamente, los resultados anteriores serán refinados en términos de operaciones elementales, para ilustrar los resultados teóricos con resultados empíricos de nuestros programas. Observemos el comportamiento general del algoritmo DIJKSTRAM, implantado con *Listas Doblemente Ligadas con Apuntador*, aplicado a una gráfica estrella $\mathcal{E}_n$.

Para **Q**, la cola de prioridades, el algoritmo requiere un número de operaciones como se indica en la Tabla 3, donde d_x es la distancia que hay entre ui y la posición correcta del dato x a insertar.

Operaciones		
Número	Nombre	Costo
n+1	**Inserta**	$O(d_x)$
n+1	**BorraMin**	$\Theta(i)$
n+1	**Vacio?**	$\Theta(1)$
1	**Principio**	$\Theta(1)$
1	**Crea_PQ**	$\Theta(1)$

Tabla 3: Número de operaciones para **Q** usando *pq_ldl*

En el mejor de los casos, la secuencia está ordenada, no hay datos que reubicar, entonces cada inserta gasta dos operaciones elementales. Por lo tanto,

$$\textbf{Koe\_PQ} = (n+1)\cdot 2 + (n+1) + (n+1) + 2 = 4(n+1) + 2 = 4n + 6.$$

$$\Longrightarrow \textbf{Koe} = (4n + 6) + n + (n+2) = 6\cdot n + 8$$

Al aplicar la implantación del DIJKSTRASORT sobre una secuencia ℓ_n organizada en forma *zig-zag-n*, se tiene: $\textbf{Koe\_PQ} = 4n + 7$ y $\textbf{KOE} = 6\cdot n + 9$, que resultan ser, también, de orden lineal, como indicamos en los análisis anteriores.

Para el peor de los casos, el algoritmo DIJKSTRAM tiene desempeño computacional de:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{n-1} t(Q.Inserta, i)$$

como podemos observar, el DIJKSTRAM sólo depende de la operación **Inserta**, la cual puede requerir tiempo $\Theta(1)$ hasta $\Theta(n)$. Lo interesante, ahora, es descubrir el tipo de secuencias para las cuales se gasta tiempo $\Theta(n)$ en cada inserción.

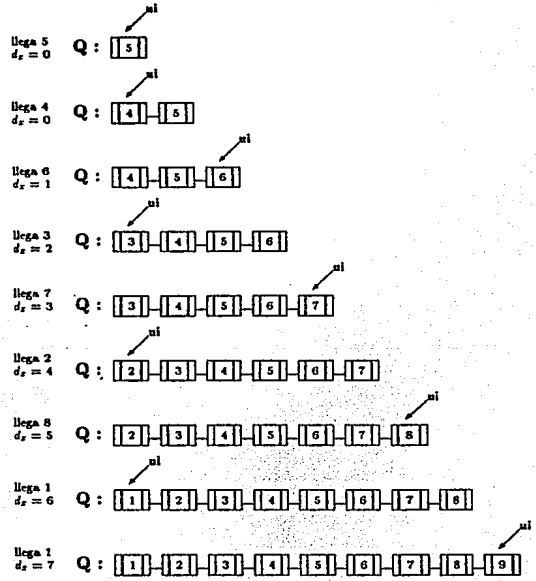


Figura 16: Ejemplo de **Inserta** aplicada a una secuencia en forma *zig-zag-1*.

Sabemos que **Inserta** depende del número de elementos que hay entre **ui** y la posición correcta del dato x a insertar, d_x. Es facil ver que: $0 \leq d_x \leq i$, donde i es el tamaño de **Q**. Así que en el peor de los casos, requerimos una secuencia de datos donde para cada inserción $d_x = \Theta(i)$. Una secuencia de datos organizada en forma zig-zag-1 tiene tal caracteristica, la Figura 16 muestra un ejemplo de esta situación, sea $\ell = \{5, 4, 6, 3, 7, 2, 8, 1, 9\}$.

Así pues, la operación inserta requiere, en total

$$\textbf{Koe\_Inserta} = \left[\begin{array}{c}\text{Numero de elementos}\\ \text{por } \textbf{Insertar} \text{ a}\\ \text{la cola } \mathbf{Q}\end{array}\right] = \left[\begin{array}{c}\textbf{Numero de saltos}\\ \textbf{para Reubicar}\\ \textbf{el dato}\end{array}\right]$$

$$\textbf{Koe\_Inserta} = (n+1) + \sum_{i=1}^{n-1} i \quad = (n+1) + \frac{(n-1)\cdot(n-2)}{2}.$$

Así que,

$$\begin{aligned}\textbf{Koe\_PQ} &= [\,(n+1) + \frac{(n-1)\cdot(n-2)}{2}\,] + (n+1) + (n+1) + 2\\ &= \frac{(n-1)\cdot(n-2)}{2} + 3\cdot n + 5.\end{aligned}$$

Luego,

$$\begin{aligned}\textbf{KOE} &= [\,\frac{(n-1)\cdot(n-2)}{2} + 3\cdot n + 5\,] + n + (n+2)\\ &= \frac{(n-1)\cdot(n-2)}{2} + 5\cdot n + 7.\end{aligned}$$

La Figura 17 nos ilustra el crecimiento de **Koe_PQ** con respecto al número de datos, n. Estos datos son el resultado de monitorear un programa con la implantación concreta de *Listas Doblemente Ligadas con Apuntador*.

Para la gráfica de la Figura 17, el eje X representa el tamaño de la secuencia a ordenar, n, el eje Y es el número de operaciones elementales, **Koe_PQ**, que requiere la cola de prioridades y "ko_pq.dj" son los **Koe_PQ**'s obtenidos al ejecutar el DIJKSTRASORT en una secuencia de datos organizada en forma zig-zag-j, $j = 1, 2, \ldots, n$.

Ahora, analizaremos el desempeño para cualquier secuencia de datos de tipo *zig-zag*. Para facilitar un poco los cálculos de las operacioes elementales, supondremos que $n = 2^t$ y $d = 2^r$, $\forall\, t, r \in \mathbf{N}$. La Figura 18 es la representación gráfica de los datos, para una secuencia de datos tipo *zig-zag-d*.

Si ℓ es una secuencia organiza en forma zig-zag-d, con $|\ell| = n = 2^t$ y $d = 2^r$, entonces el número de operaciones elementales que requiere la operación **Inserta** es:

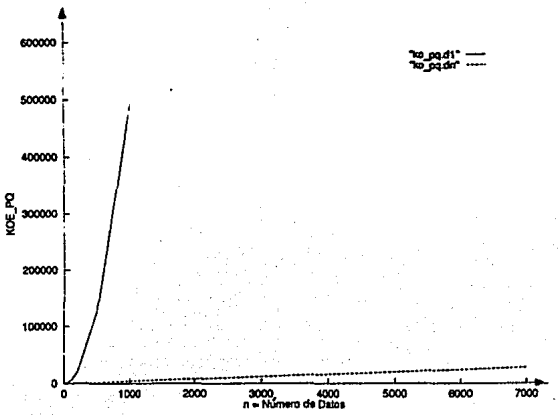


Figura 17: Comparación de **Koe_PQ**'s para ℓ_n en **zig-zag-1** y **zig-zag-n**.

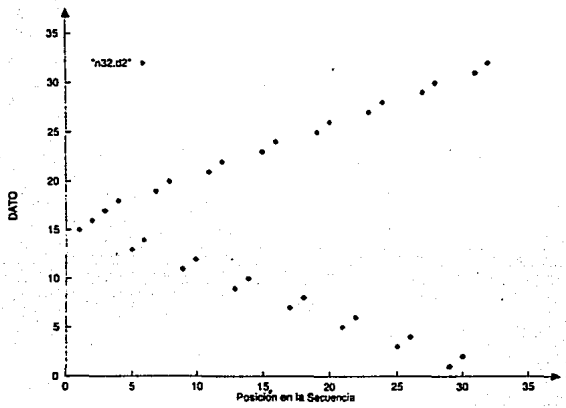


Figura 18: Gráfica para ℓ_n **zig-zag-d** con $n = 32 = 2^5$ y $d = 2 = 2^1$

$$\textbf{Koe\_Inserta} = \sum_{k=1}^{n/2d}[(2k-1)\cdot d+d] + \sum_{k=0}^{n/2d}(2k\cdot d+d)$$

$$= \sum_{k=1}^{n/2d}[(2k-1)\cdot d+d] + d + \sum_{k=1}^{n/2d}(2k\cdot d+d)$$

$$= d + \sum_{k=1}^{n/2d}[((2k-1)\cdot d+d) + (2k\cdot d+d)]$$

$$= d + \sum_{k=1}^{n/2d}[(2k\cdot d - \not{d} + \not{d} + (2k\cdot d+d)]$$

$$= d + \sum_{k=1}^{n/2d}(4k\cdot d + d) = d + \sum_{k=1}^{n/2d}(4k\cdot d) + \sum_{k=1}^{n/2d} d$$

$$= d + \frac{n}{2\not{d}}\not{d} + \sum_{k=1}^{n/2d}(4k\cdot d) = d + n + 4d\cdot\sum_{k=1}^{n/2d} k$$

$$= d + n + 4d\cdot\left(\frac{\left(\frac{n}{2d}+1\right)\cdot\left(\frac{n}{2d}\right)}{2}\right) = d + n + 4d\cdot\frac{\left(\frac{n+2d}{2d}\right)\left(\frac{n}{2d}\right)}{2}$$

$$= d + n + \not{4}\cdot\not{d}\cdot\left(\frac{(n+2d)n}{2\not{4}\cdot\not{d}\cdot d}\right) = d + n + \frac{(n+2d)\cdot n}{2d}$$

$$= d + n + \frac{n^2}{2d} + n\cdot\frac{2\not{d}}{2\not{d}} = \frac{n^2}{2d} + 2n + d$$

$$\Longrightarrow \textbf{Koe\_Inserta} = \frac{1}{2}\left[n\left(\frac{n}{d}\right)\right] + 2n + d.$$

Denotemos al número de bloques por β, entonces, tenemos que $\beta = \lceil n/d \rceil$. Podemos redefinir el número de operaciones elementales para **Inserta** en términos de β como:

$$\textbf{Koe\_Inserta} = c\cdot n\cdot\beta + O(n) + d, \quad c = 1/2$$

Nótese que si $\beta = 1$, entonces $d = n$ y la función que describe se **Koe_Inserta** es lineal en n: **Koe_Inserta** $= (1/2)\cdot n + 2n + d$, (recordemos que una lista organizada en forma zig-zag-n, es una lista ordenada en forma ascendente). Si $d = 1$ la función se vuelve cuadrática: **Koe_Inserta** $= (1/2)\cdot n^2 + 2n + d$, que resulta ser el peor caso para este tipo de datos abstracto.

La Figura 19 muestra la variación del crecimiento del contador de operaciones elementales de **PQ** para diferentes d's. Nuevamente son los valores monitoreados en nuestros programas.

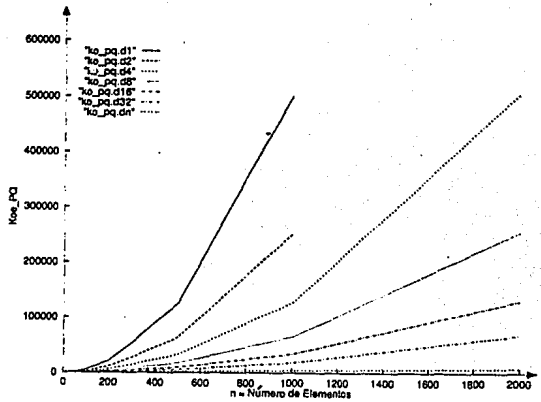


Figura 19: Comparación de **Koe_PQ** para *pq_ldl*

Para la gráfica de la Figura 19, el eje X representa el tamaño de la secuencia a ordenar, varn, el eje Y es el número de operaciones elementales que requiere la cola de prioridades, **Koe_PQ** y "ko_pq.dj" son los **Koe_PQ**'s obtenidos al ejecutar e DIJKSTRASORT en una secuencia de datos ℓ organizada en forma zig-zag- $j, = 1, 2, ..., n$.

La Figura 20 muestra la variación del crecimiento del contador de operaciones elementales para cola de prioridades **G**, para diferentes d's. Además incluye las funciones:

$$\mathbf{ko\_pq\_dn(x)} = \frac{(x-1)(x-2)}{2} + 3 + 5 \quad \text{y} \quad \mathbf{ko\_pq\_d1(x)} = 4n + 7.$$

Luego, el número total de operaciones elementales que gasta el DIJKSTRA es:

$$\mathbf{KOE} = \left[\frac{1}{2} \cdot n \cdot \beta + \Theta(n) + d\right] + (n+2) + n,$$

y en general podemos decir que: $\mathbf{KOE} = \left[\frac{1}{2} \cdot n \cdot \beta + \Theta(n)\right] + k.$

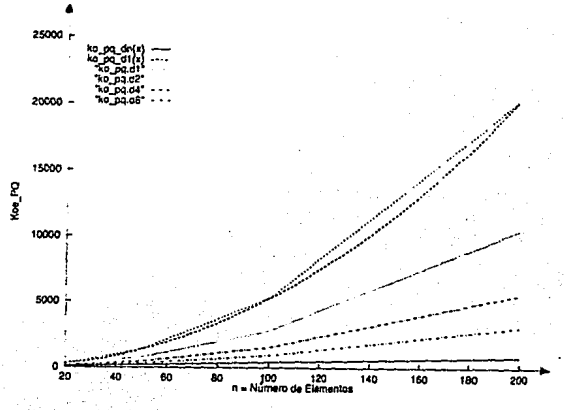


Figura 20: Comparación de **Koe_PQ** para *pq_ldl*

Nótese que si $d = 1 \implies \beta = n \implies \mathbf{KOE} = \Theta(n^2)$.

Pero, sabemos que si $d = 1$ la secuencia ℓ se encuentra muy desordenada, de hecho es el peor caso, y el número de *inversiones* es $O(n^2)$. Por lo tanto, de acuerdo con el Teorema 13 el algoritmo DIJSTRASORT es del orden $\Theta(n^2)$.

Ahora bien, si $d = 1 \implies \beta = n \implies \mathbf{KOE} = \Theta(n^2)$.

Sabemos que si $d = O(n)$ la secuencia ℓ se encuentra ordenada ascendentemente y el número de *inversiones* es $Inv^{+}(\ell) = 0$. Entonces, de acuerdo con el Teorema 13 el algoritmo DIJSTRASORT es del orden $\Theta(n)$.

Con las estas últimas observaciones volvemos a reafirmar nuestros resultados teóricos con los empíricos.

5 Level Linked Trees

En esta sección desarrollaremos el análisis para los algoritmos DIJKSTRAM y DIJKSTRASORT, utilizando, para las etiquetas temporales, la estructura de datos *Level_Linked_Trees* [23]. Empezamos dando una breve explicación de las operaciones.

Descripción de las operaciones en una lista con i elementos,

ui .- Es un apuntador a alguna de las hojas del árbol, en especial al último elemento insertado.

Inserta(x **)** .- Utilizando **Concatena**, toma tiempo $O(\log_2(\max\{1,i\}))$.

BorraMin .- Se sigue el apuntador de la raíz al mínimo elemento y se elimina. Caminamos de regreso hacia la raíz re-balanceando el árbol y actualizando el mínimo y los apuntadores sobre el camino. Requiere tiempo $O(\log_2 i)$.

EncuentraMinimo .- El mínimo se encuentra en la raíz del árbol, así que accesar al mínimo se realiza en tiempo constante.

5.1 Análisis de Adaptabilidad

Para la estructura de datos *Level_Linked_Trees*, Mehlhorn [23] demuestra los siguientes resultados

Lema 5: [23] Insertar un elemento x en un (a,b)-árbol para un conjunto varS tiene despeño computacional en el peor de los casos de $O(\log_2 |S|)$.

Lema 7: [23] Sea p un **Indicador** en T. La búsqueda de una llave k que se encuentre a d llaves de p requiere $\Theta(1+\log d)$ comparaciones.

La demostración del **Lema 5** se basa en que buscar el lugar correcto para x en una secuencia S toma tiempo $O(\log_2 |S|)$. Así que si consideramos el **Lema 7**, la búsqueda del lugar correcto toma tiempo $O(\log_2 d)$. Por lo tanto, insertar un elemento x en un LLT, con un apuntador **ui** que indique cual fue el último elemento insertado requiere $\Theta(1+\log_2 d_x)$ comparaciones, donde d_x representa la distancia entre **ui** y la posición correcta de x.

Retomando el Teorema 2 tenemos que el algoritmo DIJKSTRASORT, aplicado a una secuencia de n datos, tiene desempeño computacional en el peor de los casos de:

$$T_t(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

Pero

$$\sum_{i=1}^{nd-1} t(Q.Inserta, i) = \sum_{i=1}^{nd-1} t(Rot\_Inserta, i) + \sum_{i=1}^{nd-1} t(Compara\_Inserta, i),$$

y

$$\sum_{i=1}^{nd-1} t(Q.BorraMin, i) = \sum_{i=1}^{nd-1} t(Rot\_BMin, i) + \sum_{i=1}^{nd-1} t(Compara\_BMin, i).$$

Por lo tanto,

$$T_t(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Rot\_Inserta, i) + \sum_{i=1}^{nd-1} t(Compara\_Inserta, i)$$
$$+ \sum_{i=1}^{nd-1} t(Rot\_BMin, i) + \sum_{i=1}^{nd-1} t(Compara\_BMin, i).$$

Además, de acuerdo con Mehlhorn [23]:

$$\sum_{i=1}^{nd-1} t(Rot\_Inserta, i) + \sum_{i=1}^{nd-1} t(Rot\_BMin, i) \text{ es } \Theta(n)$$

Verificar donde se encuentra el próximo mínimo es inmediato, ya que la secuencia se encuentra en una lista ordenada, entonces: $\sum_{i=1}^{nd-1} t(Compara\_Bmin, i)$ es $\Theta(n)$.

Tenemos que, insertar un elemento x en un LLT, con un apuntador previo, **ui**, requiere $\Theta(1 + \log d)$ comparaciones. Más específicamente: $t(Q.Inserta, i) \leq \log_2 d_i$.

Pero entonces: $\sum_{i=1}^{n} \log_2 d_i = n \cdot \frac{1}{n} \sum_{i=1}^{n} \log_2 d_i$ y como la suma de los logaritmos es el logaritmo de los productos:

$$n \cdot \frac{1}{n} \sum_{i=1}^{n} \log_2 d_i = n \cdot \frac{1}{n} \log_2 \left(\prod_{i=1}^{n} d_i \right) = n \cdot \log_2 \left(\prod_{i=1}^{n} d_i \right)^{\frac{1}{n}} \leq n \cdot \log_2 \left(\frac{\sum_{i=1}^{n} d_i}{n} \right);$$

ya que la media geométrica es mayor que la media aritmética.

En la sección anterior, Teorema 13, obtuvimos que $\sum d_i \leq 2 \cdot \min\{ Inv^+(\ell), Inv^-(\ell) \}$. Por lo tanto,

$$\sum \log_2 di \leq n \cdot \log_2 \left[\frac{2 \cdot \min\{ Inv^+(\ell), Inv^-(\ell) \}}{n} \right]$$

$$\leq n \cdot \log_2 2 + n \cdot \log_2 \left[\frac{\min\{ Inv^+(\ell), Inv^-(\ell) \}}{n} \right]$$

$$\leq n + n \cdot \log_2 \left[\frac{\min\{ Inv^+(\ell), Inv^-(\ell) \}}{n} \right].$$

Por tanto,

$$T_\ell(n) = \Theta(n) + O\left(n \cdot \log_2 \left[\frac{\min\{ Inv^+(\ell), Inv^-(\ell) \}}{n} \right] \right).$$

Finalmente,

$$T_\ell(n) = \Theta(n) + O\left(n \cdot \log_2 \left[\frac{\min\{ Inv(\ell^+), Inv^-(\ell) \}}{n} \right] \right).$$

Se ha demostrado que esto coincide con la Cota Mínima [22, 8], para el problema de ORDENAMIENTO, lo que implica el siguiente resultado,

Teorema 15 *El algoritmo* DIJKSTRASORT *implantado con la estructura de datos Level_Linked_Trees es* Optimamente Adaptivo *con respecto a la medida del desorden denominada Inversiones.*

Esto implica, de acuerdo a [22, 8], que

Corolario 16 *El algoritmo* DIJKSTRASORT *es* Optimamente Adaptivo *para las medidas del desorden Max y Dis.*

6 d-Heaps

Pasamos ahora a analizar DIJKSTRAM, cuando la implantación concreta del TDA Cola de Prioridades es la estructura de datos conocida como d-Heaps. Las operaciones para un d-Heap que representa una cola con i elementos son:

siftup(x) .- Intercambia x con su predecesor mientras x no sea un nodo raíz y la llave de x sea menor que la llave de su predecesor. Requiere tiempo $O(\log_d i)$.

siftdown(x) .- Intercambia x con $MinH(x)$, el nodo con la menor llave en SUCC(x), mientras x no sea un nodo hoja y la llave de x sea mayor que la llave de $MinH(x)$. Requiere tiempo $O(d \cdot \log_d i)$.

Inserta .- El nuevo elemento x se agrega en la última posición del heap. Después se aplica el proceso **siftup(x)** Requiere tiempo $O(\log_d i)$.

EncuentraMin .- El nodo raíz del Heap contiene al elemento mínimo. Requiere tiempo constante, $\Theta(1)$.

BorraMin .- Por construcción, el nodo mínimo, x, es la raíz del Heap. Sea y el nodo almacenado en la última posición del arreglo. Se ejecuta un *Intercambio* entre x y y, después se elimina x, después se aplica un $siftdown(y)$ para restaurar el orden en el Heap. Se realiza en tiempo $O(d \cdot \log_d i)$.

Decrementa_Llave .- Decrementa la llave de x y se aplica el proceso $siftup(x)$ para restaurar el orden del Heap. Requiere tiempo $O(\log_d i)$.

6.1 Tiempos de Ejecución

Al aplicar en el Teorema 1, los tiempos de ejecución para las operaciones de la clase *pq.dh*, tenemos que el desempeño computacional del algoritmo DIJKSTRAM es:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i).$$

Pero

$$t(Q.Inserta, k) \text{ es } O(\log_d k) \Longrightarrow \sum_{i=1}^{nd-1} t(Q.Inserta, i) \text{ es } O(n \cdot \log_d n).$$

$$t(Q.BorraMin, k) \text{ es } O(d \cdot \log_d k) \Longrightarrow \sum_{i=1}^{nd-1} t(Q.BorraMin, i) \text{ es } O(n \cdot d \cdot \log_d n).$$

Entonces

$$T_{\mathcal{E}}(n) = \Theta(n) + O(n \cdot \log_d n) + O(n \cdot d \cdot \log_d n) = O(n \cdot d \cdot \log_d n).$$

De donde obtenemos los siguientes resultados:

Corolario 17 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella $\mathcal{E}_n$ e implantado con la clase pq_dh tiene desempeño computacional de*

$$T_{\mathcal{E}}(n) = O(n \cdot d \cdot \log_d n).$$

Corolario 18 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia ℓ_n de n elementos e implantado con la clase pq_dh, requiere tiempo*

$$T_{\ell}(n) = O(n \cdot d \cdot \log_d n).$$

6.2 Análisis de Adaptabilidad

El proceso **siftdown** aplicado en la operación **BorraMin** requiere de $\Theta(d \cdot \log_d i)$ operaciones, ya que al intercambiar el elemento raíz, x, con el último elemento del heap, y, que está en el último nivel del árbol (es una hoja), obliga a recorrer todos los niveles del heap, $h = O(\log_d i)$. La Figura 21 muestra un ejemplo de la operación BorraMin.

Caso I. Si ℓ está ordenada en forma ascendente.

La función **Inserta** siempre añade el nuevo elemento, x, a la última posición del Heap, como x es mayor a todos los elementos que previamente se han insertado, entonces el proceso **siftup(x)** se realiza en tiempo constante, ya que sólo verifica la condición del **While** y como ésta no se cumple termina el proceso. Por tanto: **Inserta(x)** $\sim \Theta(1)$.

Retomando el Teorema 1:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

$$\Longrightarrow T_{\mathcal{E}}(n) = \Theta(n) + \Theta(n) + \Theta(n \cdot d \cdot \log_d n) = \Theta(n \cdot d \cdot \log_d n)$$

Teorema 19 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella $\mathcal{E}_n$ e implantado con la clase pq_dh, requiere tiempo:*

$$T_{\mathcal{E}}(n) = O(n \cdot d \cdot \log_d n)$$

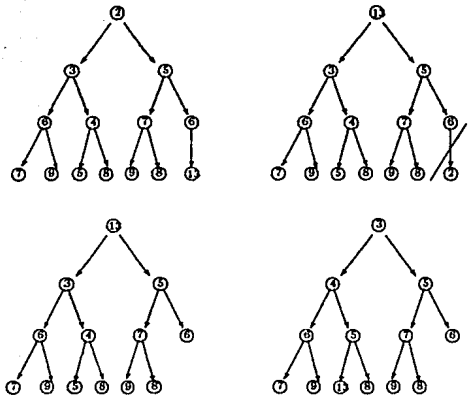


Figura 21: Ejemplo de la Operación BorraMin.

Corolario 20 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de* n *elementos,* ℓ_n *, ordenada en forma ascendente e implantado con la clase pq_dh, requiere tiempo:* $T_\ell(n) = O(n \cdot d \cdot \log_d n)$

Nótese que para el Caso I, el algoritmo DIJKSTRAM no aprovecha que la lista ya esta ordenada.

Caso II. Si ℓ está ordenada en forma descendente

Para este caso, la operación **Inserta** resulta ser un tanto ingenua, ya que agrega en la última posición del Heap a x, siendo x menor a todos los elementos antes insertados. Así que el Proceso **siftup(i)** se efectuará h veces, donde h es la altura del Heap, pero tenemos que $h = O(\log_d n)$, entonces, **Inserta** requiere tiempo $O(\log_d n)$.

Teorema 21 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de* n *elementos, ordenada en forma descendente e implantado con la clase pq_dh, requiere tiempo:* $T_E(n) = \Theta(n \cdot d \cdot \log_d n)$.

Por lo tanto, para el caso II, el algoritmo requiere el tiempo calculado para el peor de los casos.

Caso III. Si ℓ ha sido obtenida aletoriamente bajo una distribución uniforme.

El comportamiento esperado del algortimo DIJKSTRAM es:

$$\begin{aligned}
\mathbf{E}[\mathcal{T}_{\mathcal{E}}(n)] &= \mathbf{E}\left[\Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)\right] \\
&= \mathbf{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathbf{E}[t(Q.Inserta, i)] + \sum_{i=1}^{nd} \mathbf{E}[t(Q.BorraMin, i)] \\
&= \Theta(n) + \Theta(n \cdot \log_d i) + \Theta(n \cdot d \cdot \log_d i) = \Theta(n \cdot d \cdot \log_d i).
\end{aligned}$$

Entonces obtenemos el siguiente resultado,

Teorema 22 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia ℓ de n elementos, obtenida aleatoriamente, e implantado con la clase pq_dh, requiere tiempo:* $\mathcal{T}_{\ell}(n) = \Theta(n \cdot d \cdot \log_d n)$.

Resulta ser el mismo desempeño computacional que se tiene para esta implantación en el peor de los casos. Por tanto obtenemos el siguiente resultado,

Corolario 23 *El algoritmo* DIJKSTRASORT *aplicado a cualquier secuencia de datos ℓ no es adaptivo.*

Los *Heap Binarios* son un caso particular de los *d-Heaps*, para $d = 2$. Por tanto, considerando los resultados anteriores con, $d = 2$ se tiene que el desempeño computacional del algoritmo DIJKSTRAM, aplicado a graficas estrella $\mathcal{E}_n$, en el peor de los casos, es $\Theta(n \cdot \log_2 n)$.

7 Colas Binomiales

Ahora, pasaremos a revisar la estructura de datos denominada, *Colas Binomiales*, iniciaremos dando una breve descripción de las Operaciones para una cola de prioridades Q con i elementos, utilizando la clase *pq_bq*, descrita en el Apéndice **B**:

MinH .- Apuntador al Nodo Mínimo.

Funde .- (CB: Cola_Binomial) [Meld] Une CB con **Q**. Une los elementos de CB y **Q**, mientra existan arboles binomiales con igual indice en **Q**. Requiere tiempo $O(\log t)$, donde t es el número de elementos en la cola resultante.

Inserta(x) .- Crea una cola binomial CB consistente de un sólo objeto, el nodo x. Despues efectúa **Funde** CB con **Q**. Requiere tiempo $O(\log i)$.

EncuentraMin .- Regresa el "valor" de $MinH$. Requiere tiempo constante, $\Theta(1)$.

BorraMin .- Primero se asigna $nmin \leftarrow MinH$. Después elimina el nodo mínimo, esto incrementa el número de arboles en la cola, los cuales hay que reorganizar aplicando **funde**, finalmente regresa $nmin$. Requiere tiempo $O(\log i)$.

Decrementa_LLave(x,k) .- Decrementa el valor de la llave de x a k y después aplica el proceso $Siftup(x)$ para restaurar el orden del árbol. Requiere tiempo $O(\log i)$.

Siftup(x) .- Intercambia x con su predecesor mientras x no sea un nodo raíz y la llave de x que la llave de su predecesor. Requiere tiempo $O(\log i)$.

7.1 Tiempos de Ejecución

El desempeño computacional del algoritmo DIJKSTRAM utilizando la *Clase* *pq_bq*, de acuerdo al Teorema 1 es:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i).$$

Tenemos que:

$$t(Q.BorraMin, k)) \ es \ O(\log k) \implies \sum_{i=1}^{nd-1} t(Q.BorraMin, i) \ es \ O(n \cdot \log n).$$

$$t(Q.Inserta, k) \ es \ O(\log k) \implies \sum_{i=1}^{nd-1} t(Q.Inserta, i) \ es \ O(n \cdot \log n).$$

Entonces,

$$T_{\mathcal{E}}(n) = \Theta(n) + O(n \cdot \log_2 n) + O(n \cdot \log_2 n) = O(n \cdot \log_2 n).$$

Corolario 24 *El Algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *e implantando con la Clase pq_bq tiene desempeño computacional, en el peor de los casos, de:* $T_{\mathcal{E}}(n) = O(n \cdot \log_2 n)$.

7.2 Análisis de Adaptabilidad

Caso I. Si ℓ está ordenada en forma ascendente

La operación **Inserta** va construyendo los arboles binomiales independientemente de que la cola ya este ordenada o no, por tanto su desempeño computacional es $\Theta(log_2 n)$. Aunque **Inserta** deje el *Nodo Mínimo* en el árbol más pequeño, la operación **BorraMin** debe restaurar **MinH**, lo cual requiere tiempo $\Theta(\log_2 n)$.

Por lo cual podemos concluir los siguientes resultados:

Teorema 25 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *e implantado con la clase pq_bq, requiere tiempo:* $T_\ell(n) = \Theta(n \cdot \log_2 n)$.

Corolario 26 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia* ℓ *de* n *elementos, ordenada en forma ascendente e implantado con la clase pq_bq, requiere tiempo:* $T_\ell(n) = \Theta(n \cdot \log_2 n)$.

Así pues, para el Caso I, el algoritmo DIJKSTRASORT no aprovecha que la lista ya esta ordenada.

Caso II. Si ℓ está ordenada en forma descendente

Para este caso, la operación **Inserta** deja el *nodo mínimo* en el árbol más grande, con mayor rango, la operación **BorraMin** debe realizar varios procesos **Meld** para reorganizar la cola binomial, por lo tanto la operación **BorraMin** requieren tiempo $\Theta(\log_2 n)$.

Entonces, obtenemos el siguiente resultado.

Teorema 27 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de* n *elementos, ordenada en forma descendente e implantado con la clase pq_dh, requiere tiempo:* $T_\ell(n) = \Theta(n \cdot \log_2 n)$.

Caso III. ℓ ha sido obtenida aletoriamente bajo una distribución uniforme.

El comportamiento esperado del algortimo es:

$$\begin{aligned}
\mathbf{E}[T_\ell(n)] &= \mathbf{E}\left[\Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta,i) + \sum_{i=1}^{nd} t(Q.BorraMin,i)\right] \\
&= \mathbf{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathbf{E}[t(Q.Inserta,i)] + \sum_{i=1}^{nd} \mathbf{E}[t(Q.BorraMin,i)] \\
&= \Theta(n) + \sum_{i=1}^{nd-1} O(\log_2 i) + \sum_{i=1}^{nd} O(\log_2 i) \\
&= \Theta(n) + \Theta(n \cdot \log_2 n) = \Theta(n \cdot \log_2 n).
\end{aligned}$$

De donde obtenemos el siguiente resultado,

Teorema 28 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de n elementos, obtenida aleatoriamente, e implantado con la clase pq_dh, requiere tiempo:* $T_\ell(n) = O(n \cdot \log_2 n)$.

Para los tres casos anteriores se ha obtenido el mismo desempeño computacional que se tiene para esta implantación en el peor de los casos. Por tanto, obtenemos el siguiente resultado.

Corolario 29 *El algoritmo* DIJKSTRASORT *aplicado a cualquier secuencia de datos ℓ no es adaptivo.*

8 Fibonacci Heaps

Realizaremos ahora, el análisis de los algoritmos DIJKSTRASORT y DIJKSTRAM para la estructura de datos llamada *Fibonacci Heaps (F-Heaps).* Iniciamos dando una breve descripción de las Operaciones para una cola de prioridades, Q, con i elementos, utilizando la clase *pq_fh*, definida en el Apéndice B.

Inserta .- Agrega un nuevo elemento al Heap. Requiere tiempo $\Theta(1)$.

EncuentraMin .- Regresa el nodo cuya llave es la mínima del Heap. Por construcción, el **F-Heap** tiene un apuntador al *Nodo Mínimo*, por tanto esta operación gasta tiempo $\Theta(1)$.

BorraMin .- Almacena en una variable auxiliar el *Nodo Mínimo*, lo elimina de la lista de raices, integra los hijos del *Nodo Mínimo* a la lista de raices, finalmente, 'reorganiza' los arboles del F-Heap, utilizando la operación **Link**. Gasta en total tiempo amortizado de $O(\log_2 i)$.

DecrementaLlave(i, Δ) .- Decrementa la llave de i en Δ unidades. Requiere desempeño computacional amortizado $\Theta(1)$.

8.1 Tiempos de Ejecución

Al aplicar Teorema 1 obtenemos que el desempeño computacional del DIJKSTRAM utilizando la *Clase pq_fh* es:

$$T_{\mathcal{E}}(n) = \Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta.i) + \sum_{i=1}^{nd} t(Q.BorraMin,i).$$

Tenemos que:

$$t(Q.BorraMin.k)) \ es \ O(\log k) \implies \sum_{i=1}^{nd-1} t(Q.BorraMin.i) \ es \ O(n \cdot \log n).$$

$$t(Q.Inserta.k) \ es \ O(1) \implies \sum_{i=1}^{nd-1} t(Q.Inserta.i) \ es \ O(n).$$

Entonces.

$$T_{\mathcal{E}}(n) = \Theta(n) + O(n) + O(n \cdot \log_2 n) = O(n \cdot \log_2 n).$$

Por lo tanto, obtenemos los siguientes resultados:

Teorema 30 *El Algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}$ *e implantando con la Clase pq_fh tiene desempeño computacional, en el peor de los casos, de:* $T_{\mathcal{E}}(n) = O(n \cdot \log_2 n)$.

Teorema 31 *El Algoritmo* DIJKSTRASORT *aplicado a una secuencia* ℓ *de* n *datos e implantando con la Clase pq_fh tiene desempeño computacional, en el peor de los casos, de:* $T_{\mathcal{E}}(n) = O(n \cdot \log_2 n)$.

8.2 Análisis de Adaptabilidad

Caso I. Si ℓ está ordenada en forma ascendente.

La operación **BorraMin** debe restaurar el F-Heap, auxiliandose del arreglo de apuntadores, que es de tamaño $\lfloor \cdot$. lo $\cdot$ l requiere tiempo $\Theta(\log_2 n)$.

Por lo cual podemos concluir los siguientes resultados:

Teorema 32 *El algoritmo* DIJKSTRAM *aplicado a una gráfica estrella* $\mathcal{E}_n$ *e implantado con la clase pq_fh, requiere tiempo:* $T_\ell(n) = O(n \cdot \log_2 n)$.

Corolario 33 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia* ℓ *de* n *elementos, ordenada en forma ascendente e implantado con la clase pq_fh, requiere tiempo:* $T_\ell(n) = O(n \cdot \log_2 n)$.

Así pues, para el Caso I, el algoritmo DIJKSTRASORT no aprovecha que la lista ya esta ordenada.

Caso II. Si ℓ está ordenada en forma descendente.

De manera similar al caso anterior, concluimos:

Teorema 34 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de* n *elementos, ordenada en forma descendente e implantado con la clase pq_fh, requiere tiempo:* $T_\ell(n) = O(n \cdot \log_2 n)$.

Caso III Si ℓ ha sido obtenida aletoriamente bajo una distribución uniforme.

El comportamiento esperado del algortimo DIJKSTRAM es:

$$\begin{aligned}
\mathbf{E}[T_\ell(n)] &= \mathbf{E}\left[\Theta(n) + \sum_{i=1}^{nd-1} t(Q.Inserta, i) + \sum_{i=1}^{nd} t(Q.BorraMin, i)\right] \\
&= \mathbf{E}[\Theta(n)] + \sum_{i=1}^{nd-1} \mathbf{E}[t(Q.Inserta, i)] + \sum_{i=1}^{nd} \mathbf{E}[t(Q.BorraMin, i)] \\
&= \Theta(n) + \sum_{i=1}^{nd-1} \Theta(1) + \sum_{i=1}^{nd} \Theta(\log_2 i) \\
&= \Theta(n) + \Theta(n) + \Theta(n \cdot \log_2 n).
\end{aligned}$$

Por lo tanto concluimos el siguiente resultado:

Teorema 35 *El algoritmo* DIJKSTRASORT *aplicado a una secuencia de n elementos, obtenida aleatoriamente e implantado con la clase pq_fh, requiere tiempo:*

$$T_\ell(n) = \Theta(n \cdot \log_2 n).$$

Para los tres casos anteriores se ha obtenido el mismo desempeño computacional que se tiene para esta implantación en el peor de los casos. Por tanto concluimos los siguientes resultados:

Corolario 36 *El algoritmo* DIJKSTRASORT *aplicado a cualquier secuencia de datos ℓ de tamaño n tiene un desempeño computacional de* $\Theta(n \cdot \log_2 n)$.

Corolario 37 *El algoritmo* DIJKSTRASORT *aplicado a cualquier secuencia de datos ℓ de tamaño n no es adaptivo.*

8.3 Resultados Empíricos

A continuación refinaremos, en términos de operaciones elementales los resultados teóricos obtenidos anteriormente ilustrandolos con los resultados empíricos de nuestros programas.

De acuerdo con el Teorema 1 el DIJKSTRAM tiene un desempeño computacional

$$T_\mathcal{E}(n) = \Theta(n) + \sum_{i=1}^{nd} t(Q.BorraMin, i)$$

es decir, depende específicamente de la operación **BorraMin**, la cual se describe esquemáticamente en la Figura 22.

Las operaciones 1, 2, 5 y 7 gastan tiempo constante. La operación 3 depende del número de hijos que tenga el NodoMinimo, ya que primero hay que desligar a cada nodo de su padre, pues pasan a ser raices y unir la lista de hijos de NodoMinimo a la lista de raices se efectua en tiempo constante. Para la operación 4 tenemos que cada **Link** gasta tiempo constante, pero en el peor caso se tienen $(\log i)$ árboles en **Q**, por tanto esta operación es $O(\log i)$. Para la operación 6, se debe recorrer todo el arreglo de apuntadores, el cual es de tamaño $(\log i)$, por tanto la operación es $\Theta(\log i)$.

BorraMin
InicioProceso
1. MinAux ← NodoMinimo
2. Elimina NodoMinimo de la lista de raices
3. **Si** NodoMinimo tiene Hijos **Entonces**
   Agrega los hijos del NodoMinimo a la lista de raices
   Fin-Si
4. Almacena los arboles del F-Heap, en el arreglo y
   los va ligando como van llegando al arreglo
   si tienen el mismo rank.
5. Reinicializa **Q**.
6. Vacea el contenido del arreglo en **Q**.
7. Regresa el MinAux

FinProceso

Figura 22: Proceso **BorraMin**

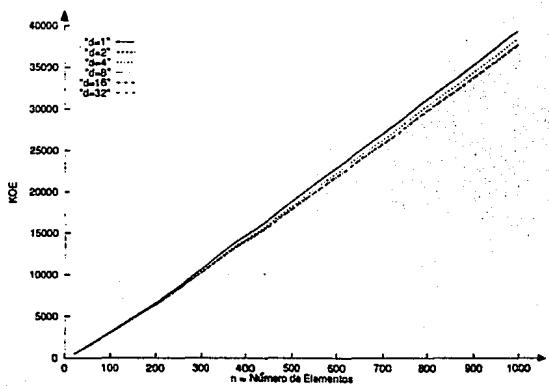


Figura 23: Comparación de **KOE** para *pq-FH*

Finalmente, tenemos que $T_E(n) = \Theta(n \cdot \log_2 n)$ y por tanto reconfirmamos que el algoritmo no es adaptivo.

La gráfica 23 muestra el desempeño computacional del DIJKSTRASORT aplicado a listas de datos organizadas en forma *xig-zag-d*. El eje **X** representa el tamaño de la lista, varn, y el eje **Y** el número de operaciones elementales, **KOE**, que gasto el algoritmo. Como podemos observar, crecimiento del número de operaciones elementales, con respecto a el numero de datos, varía sólo en una constante.

VI Otras evidencias de Adaptabilidad en RMC

1 Rutas

Consideremos la gráfica de la forma:

$$\mathcal{L}: \quad v_o, (v_o, v_1), v_1, (v_2, v_3), v_3, \ldots, v_{n-1}, (v_{n-1}, v_n), v_n.$$

es decir, esta gráfica ya es un camino, ver Figura 24. Si la gráfica tiene n aristas, diremos que se trata de una $\mathcal{L}_n$.

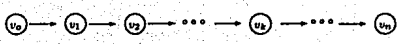


Figura 24: Ruta $\mathcal{L}_n$.

Afirmación 38 *Sea* **Q** *la cola de prioridades que utiliza el algoritmo de* DIJKSTRA *para almacenar las etiquetas temporales finitas. Supongamos que* **G** *se almacena como una lista de adyacencia. Si* **G** *es una gráfica* $\mathcal{L}_n$, *entonces al aplicar el* DIJKSTRA *sobre* **G** *se tiene que, en todo momento, el número de elementos en* **Q** *es a lo más uno.*

Justificación.

Sea **G** una gráfica $\mathcal{L}_n$ para la cual c_j representa el costo del arco $(j-1, j)$, $\forall\ 1 \leq j \leq n$.

Tenemos que **G** es de la forma:

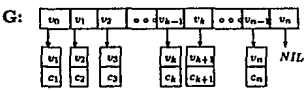


Figura 25: Lista de Adyacencias para la Gráfica **G**

Siguiendo el algoritmo de DIJKSTRA,

$\mathbf{Q}$: $[\, v_o, 0 \,]$ *borra* v_o $\Longrightarrow \mathbf{Q} = \phi$.

Revisa los vecinos de v_o, es sólo uno: $[\, v_1, c_1 \,]$ con etiqueta infinito;
modifica la etiqueta e inserta v_1 en $\mathbf{Q}$.

$\mathbf{Q}$: $[\, v_1, c_1 \,]$ *borra* v_1 $\Longrightarrow \mathbf{Q} = \phi$.

Revisa los vecinos de v_1, es sólo uno: $[\, v_2, c_2 \,]$ con etiqueta infinito;
modifica la etiqueta e inserta v_2 en $\mathbf{Q}$.

...

En el k-ésimo vértice:

$\mathbf{Q}$: $[\, v_k, \sum_{j=1}^{k}(c_j) \,]$ *borra* v_k $\Longrightarrow \mathbf{Q} = \phi$.

Revisa los vecinos de v_k, es sólo uno: $[\, v_{k+1}, c_{k+1} \,]$ con etiqueta infinito, entonces
modifica la etiqueta e inserta v_{k+1} en $\mathbf{Q}$.

...

En el $(n-1)$-ésimo vértice:

$\mathbf{Q}$: $[\, v_{(n-1)}, \sum_{j=1}^{n-1}(c_j) \,]$ *borra* $v_{(n-1)}$ $\Longrightarrow \mathbf{Q} = \varphi$.

Revisa los vecinos de $v_{(n-1)}$, es sólo uno: $[\, v_n, c_n \,]$ con etiqueta infinito;
modifica la etiqueta e inserta v_n en $\mathbf{Q}$.

En el n-ésimo vértice: $\mathbf{Q}$: $[\, v_n, \sum_{j=1}^{n}(c_j) \,]$ *borra* v_n $\Longrightarrow \mathbf{Q} = \phi$.

Revisa los vecinos de v_n: no tiene, entonces termina el algortimo.

$\mathcal{L}_n$ representa una familia de gráficas, para las cuales, la cola de prioridades tiene a lo más un elemento luego, para cada uno de lo TDC anteriores las operaciones se realizan en tiempo constante. Observemos ahora, el comportamiento de las implantaciones concretas del DIJKSTRA aplicado a esta familia de gráficas.

Listas Simplemente Ligadas.

Para la clase *pq_lsl*, tenemos que cada **BorraMin** realiza (i+4) operaciones elementales, pero i es a lo más uno, entonces:

$$\mathbf{Koe\_PQ} = (n+1)\,\mathbf{BorraMin} + (n+1)\,\mathbf{Inserta} + (n+1)\,\mathbf{Vacio} + 1\cdot\mathbf{Principio} + 1\cdot\mathbf{CreaPQ}.$$

$$\mathbf{Koe\_PQ} \leq 5\cdot(n+1) + 2\cdot(n+1) + 1\cdot(n+1) + 2 \leq 8\cdot n + 10.$$

$$\Longrightarrow \mathbf{KOE} \leq (8\cdot n + 10) + (n+2) + n \leq (10\cdot n + 12) = \Theta(n).$$

Listas Doblemente Ligadas.

Como **Q** tiene a lo más un elemento, cuando se inserta un nuevo dato, se hace la operación sobre una cola vacía, por lo que cada inserta gasta tiempo constante en efectuarse. Por tanto: **Koe_PQ** es $\Theta(n) \Rightarrow$ **KOE** es $\Theta(n)$.

Fibonacci Heaps.

Ya que la cola de prioridades **Q** tiene a lo más un elemento, borrar el mínimo en un lista de un dato, gasta tiempo constante. Por tanto: **Koe_PQ** es $\Theta(n)$ y **KOE** es $\Theta(n)$.

La figura 26 ilustra gráficamente el comportamiento de las implantaciones concretas aplicadas sobre la familia $\mathcal{L}_n$. El eje **X** representa la variación en n y el eje **Y** el número de operaciones elementales, **KOE.**

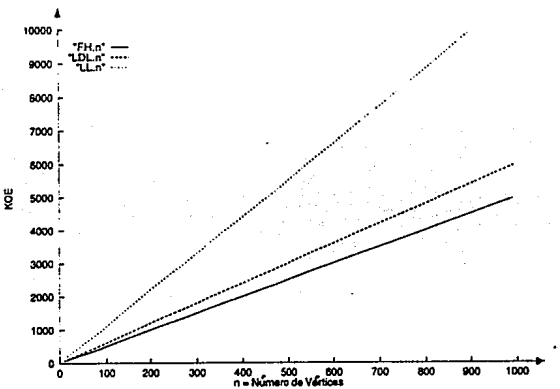


Figura 26: DIJSKTRA aplicado a $\mathcal{L}_n$.

Con el análisis previo, hemos demostrado el siguiente resultado:

Teorema 39 *El Algoritmo de* DIJKSTRA *aplicado a una gráfica* $\mathcal{L}_n$ *e implantado con las clases pq_lsl, pq_ldl y pq_fh requiere tiempo* $\Theta(n)$.

De hecho, tenemos que:

Teorema 40 *Sea* **G** *una gráfica* $\mathcal{L}_n$ *y sea* **Q** *la estructura donde se almacenen las etiquetas temporales finitas que genera el algoritmo de* DIJKSTRA. *Cualquier implantación concreta del* DIJKSTRA, *en el modelo computacional de comparaciones, que represente a* **G** *como una lista de adyacencias y a* **Q** *como una cola de prioridades, tiene un desempeño computacional de* $\Theta(n)$.

Justificación.

Como la cola de prioridades **G** siempre tiene a lo más un elemento y, en el modelo computacional de comparaciones, **insertar** o **borrar** un elemento en una estructura de tamaño menor o igual que uno gasta tiempo constante. Entonces la ejecución total del algoritmo, requiere tiempo lineal.

2 Árboles T_{kh}

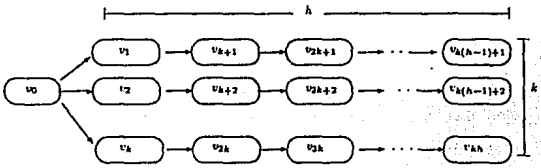


Figura 27: Gráfica T_{kh}

Definición.

Sea T_{kh} un árbol cuya raiz tiene k hijos, todos los demás nodos tienen un sólo hijo y cada rama tiene profundidad h.

La Figura 27 representa un árbol T_{kh} y la Figura 28 representa la lista de adyacencias.

Afirmación 41 *Sea* **Q** *la cola de prioridades que utiliza el algoritmo de* DIJKSTRA *para almacenar las etiquetas temporales finitas. Supongamos que* **G** *se almacena como una lista de adyacencia. Si* **G** *es una gráfica* T_{kh}, *entonces al aplicar el* DIJKSTRA *sobre* **G** *se tiene que el número de elementos en* **Q** *es a lo más* k.

$G = T_{kh}$:

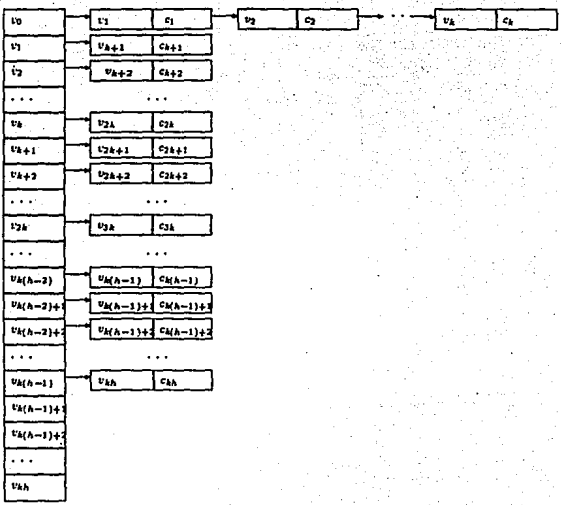


Figura 28: Lista de Adyacencias de la Gráfica T_{kh} .

Justificación.

Sea **G** una gráfica T_{kh} para la cual c_j representa el costo del arco $(j, j+k)$ para todo j tal que $0 \leq j \leq k(h-1)$.

Siguiendo el algoritmo de DIJKSTRA.

Q: $[\, v_0, 0 \,]$ *borra* v_0 $\Longrightarrow$ **Q** = o.

Revisa los vecinos de v_0, son k: $[\, v_1, c_1 \,]$, $[\, v_2, c_2 \,]$, ... ,$[\, v_k, c_k \,]$ todos con etiqueta infinito; modifica la etiqueta e inserta v_1 ... v_k en **Q**.

$\Longrightarrow$ **Q**: $[\, v_1, c_1 \,]$-$[\, v_2, c_2 \,]$- ... -$[\, v_k, c_k \,]$

borra el mínimo. supongamos, sin pérdida de generalidad, que el mínimo es v_1

$\Longrightarrow$ **Q**: $[\, v_2, c_2 \,]$- ... -$[\, v_k, c_k \,]$.

Revisa los vecinos de v_1, es sólo uno: $[\, v_{k+1}, c_{(k+1)} \,]$ y tiene etiqueta infinito: modifica la etiqueta y lo inserta en **Q**.

Nótese que, como cada vértice v_k tiene sólo un vecino, $\forall\ 1 \leq k < n$, al borrar el vértice v_k se agrega a la cola únicamente un dato. Es decir, cada vez que se borra un dato, éste es reemplazado por su vecino, si lo tiene, y si no tiene vecino la cola de prioridades disminuye en uno su tamaño. Por tanto, G tiene a lo más k elementos.

Las gráficas T_{kh} representan una familia de gráficas, para las cuales, la cola de prioridades tiene a lo más k elementos y además el número total de vértices en la gráfica es $n = k \cdot h + 1$. esto es. n es $\Theta(k \cdot h)$. Observamos que, en especial, las gráfias $\mathcal{L}_n$ son T_{1n}.

Ahora. analicemos el comportamiento de las implantaciones concretas del algoritmo DIJKSTRA aplicado a la familia de gráficas T_{kh}.

Listas Simplemente Ligadas.

Tenemos que cada **BorraMin** realiza (i+4) operaciones elementales, pero i es a lo más k, entonces:

$$\textbf{Koe\_PQ} = (n+1)\,\textbf{BorraMin} + (n+1)\,\textbf{Inserta} + (n+1)\,\textbf{Vacio} + 1\cdot\textbf{Principio} + 1\cdot\textbf{CreaPQ}$$

$$\textbf{Koe\_PQ} \leq (k+4)(n+1) + 3\cdot(n+1) + 2 \leq (k+7)\cdot(n+1) + 2.$$

$$\Longrightarrow \textbf{KOE} \leq ((k+7)\cdot(n+1)) + (n+2) + n \leq (k+9)\cdot(n+1).$$

pero $n = k \cdot h + 1$.

$$\Longrightarrow \textbf{KOE} \leq (k+9)\cdot(k\cdot h+2) \leq (k\cdot h)\cdot(k+9) + 2\cdot(k+9).$$

Nótese que si $h = 1$ la función del lado derecho de la desigualdad, se vuelve cuadrática en k, que es el caso de las gráficas estrella analizadas en el capítulo anterior, ya que una gráfica $\mathcal{E}_k$ es una T_{k1}. Así que, **KOE** $= \Theta(n)$ siempre y cuando $h >> k$.

Listas Doblemente Ligadas.

Como **Q** tiene a lo más k elementos, cuando se inserta un nuevo dato, se hace la operación sobre una cola de tamaño $k-1$, por lo que cada inserción se efectúa, en el peor de los casos, en tiempo $O(k)$. Cuando $h = 1$ el desempeño computacional del algoritmo es de orden cuadrático, pero si h tiende a ser más grande que k, esto es $h >> k$, el algoritmo será lineal en n. Formalmente,

Koe_PQ es $\Theta(n) \Rightarrow$ **KOE** es $\Theta(n)$ siempre y cuando $h >> k$.

Fibonacci Heaps.

Ya que la gráfica **Q** tiene a lo más k elementos, borrar el mínimo en una lista de k datos, gasta tiempo $O(\log_2 k)$, que tiende a ser constante cuando $h >> k$ Por tanto:

Koe_PQ es $\Theta(n) \Rightarrow$ **KOE** es $\Theta(n)$ siempre y cuando $h >> k$.

Las Figuras 29,30,31 ilustran gráficamente el comportamiento de las implantaciones concretas aplicadas sobre la familia T_{kh}, para los TDC, *pq_lsl, pq_ldl, pq_fh*, respectivamente El eje **X**, en cada una de las Figuras, representa la variación en n y el eje **Y** el número de operaciones elementales, **KOE**.

Teorema 42 *El Algoritmo de* DIJKSTRA *aplicado a una gráfica* T_{kh} *e implantado con las clases pq_lsl, pq_ldl y pq_fh requiere tiempo* $\Theta(n)$, *siempre que* $h >> k$.

De hecho, tenemos que:

Teorema 43 *Sea* **G** *una gráfica* T_{kh} *y sea* **Q** *la estructura donde se almacenen las etiquetas temporales finitas que genera el Algoritmo de* DIJKSTRA.
El Algoritmo de DIJKSTRA *aplicado a una gráfica* T_{kh} *requiere tiempo* $\Theta(n)$, *siempre que* $h >> k$.

Sólo para ejemplificar, analicemos con detalle las gráficas T_{2h} y T_{3h}.

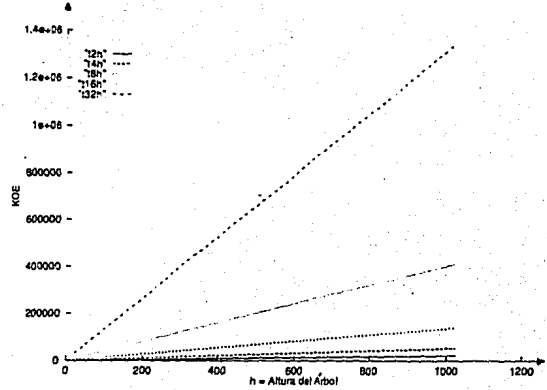


Figura 29: DIJSKTRA aplicado a T_{kh} usando pq_lsl

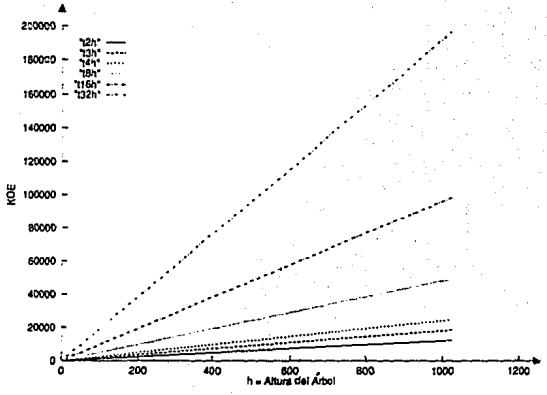


Figura 30: DIJSKTRA aplicado a T_{kh} usando pq_ldl

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

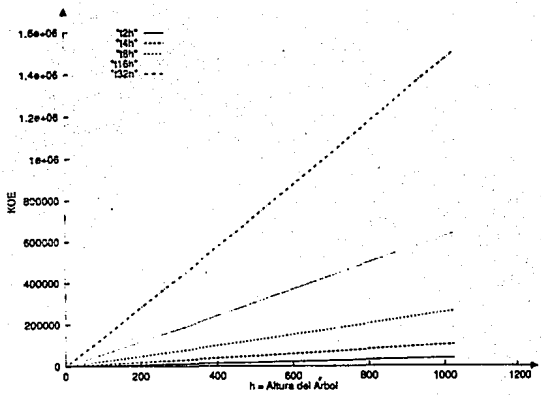


Figura 31: DIJSKTRA aplicado a T_{kh} usando pq_fh

Árboles T_{2h}

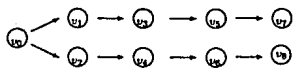


Figura 32: gráfica $T_{2,4}$

Ahora, consideremos la gráfica de la Figura 32, ésta es un árbol cuya raiz tiene dos hijos y cada rama tiene profundidad cuatro, es decir es una gráfica T_{24} La Figura 33 representa las listas de adyacencia para las gráficas T_{2h} y T_{24}.

Las gráficas T_{2h} representan una familia de gráficas, para las cuales, la cola de prioridades, **Q**, tiene a lo más dos elementos, para cada uno de los TDC anteriores, las operaciones se realizan en tiempo constante. Observemos, detalladamente, el comportamiento de las implantaciones concretas del algoritmo DIJKSTRA aplicado a esta familia.

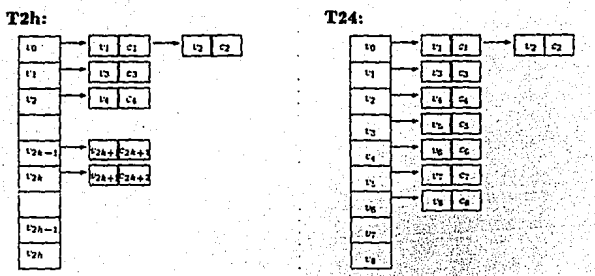


Figura 33: Listas de Adyacencia para T_{2h} y T_{24}.

Listas Simplemente Ligadas

Tenemos que cada **BorraMin** realiza (i+4) operaciones elementales, pero i es a lo más dos. entonces:

$$\begin{aligned}\textbf{Koe\_PQ} = {} & (n+1)\,\textbf{BorraMin} + (n+1)\,\textbf{Inserta} + (n+1)\,\textbf{Vacio} \\ & + 1\cdot\textbf{Principio} + 1\cdot\textbf{CreaPQ}\end{aligned}$$

$$\textbf{Koe\_PQ} \leq 6\cdot(n+1)+3\cdot(n+1)+2 \leq 9\cdot n + 11.$$

$$\Longrightarrow \textbf{KOE} \leq (9\cdot n+11)+(n+2)+n \leq (11\cdot n+13) = \Theta(n).$$

Listas Doblemente Ligadas

Como **Q** tiene a lo más dos elementos, cuando se inserta un nuevo dato, se hace la operación sobre una cola de tamaño uno, por lo que cada inserta se efectua en tiempo constante Por tanto: **Koe_PQ** es $\Theta(n) \Rightarrow$ **KOE** es $\Theta(n)$.

Fibonacci Heaps

Como **Q** tiene a lo más dos elementos, borrar el minimo en una lista de tamaño dos. requiere un mumero constante, de operaciones elementales. Por tanto:

$$\textbf{Koe\_PQ} \text{ es } \Theta(n) \Rightarrow \textbf{KOE} \text{ es } \Theta(n).$$

Árboles T_{3h}

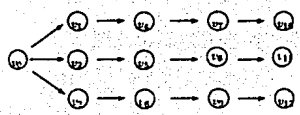


Figura 34: Gráfica $T_{3,4}$.

Consideremos la gráfica de la figura 34 que representa a la gráfica $T_{3,4}$, ésta es un árbol cuya raiz tiene 3 hijos y cada rama tiene profundidad 4. La Figura 35 representa la lista de adyacenciaspara esta gráfica.

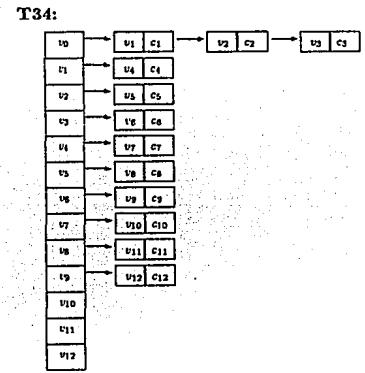


Figura 35: Lista de Adyacencias de la Gráfica $T_{3,4}$.

Las gráficas $T_{3,h}$ representan una familia de gráficas, para las cuales, la cola de prioridades tiene a lo más tres elementos. observemos el comportamiento de las implantaciones concretas del DIJKSTRA aplicado a esta familia.

Listas Simplemente Ligadas.

Tenemos que cada **BorraMin** realiza (i+4) operaciones elementales, pero i es a lo más tres, entonces:

$$\begin{aligned}\textbf{Koe\_PQ} = {} & (n+1)\,\textbf{BorraMin} + (n+1)\,\textbf{Inserta} + (n+1)\,\textbf{Vacio} \\ & + 1\cdot\textbf{Principio} + 1\cdot\textbf{CreaPQ}\end{aligned}$$

$$\textbf{Koe\_PQ} \leq 7\cdot(n+1)+3\cdot(n+1)+2 \leq 10\cdot n + 12.$$

$$\Longrightarrow \textbf{KOE} \leq (10\cdot n+11)+(n+2)+n \leq (12\cdot n+14) = \Theta(n).$$

Listas Doblemente Ligadas.

Como Q tiene a lo más tres elementos, cuando se inserta un nuevo dato, se hace la operación sobre una cola de tamaño dos, por lo que cada inserta se efectúa en tiempo constante Por tanto: **Koe_PQ** es $\Theta(n)$ $\Rightarrow$ **KOE** es $\Theta(n)$.

Fibonacci Heaps.

Como Q tiene a lo más tres elementos, borrar el mínimo en una lista de tres datos requiere un número constante de operaciones elementales. Por tanto: **Koe_PQ** y **KOE** son $\Theta(n)$.

La Figura 36 ilustra gráficamente el comportamiento de las implantaciones concretas: *pq_lsl, pq_ldl, pq_fh*, aplicadas sobre la familia de gráficas T_{2h}. El eje **X** representa la variación en n y el eje **Y** el número de operaciones elementales, **KOE**.

3 d-Árboles

Un **d-árbol** es un árbol balanceado para el cual cada nodo tiene d hijos, a excepción de las hojas, las cuales no tienen hijos. La Figura 37 representa un d-árbol, para $d = 3$. Nótese que si **G** es un d-árbol con profundidad h, entonces el número total de nodos en **G** es: $n = \sum_{j=0}^{h} d^j = \frac{d^{h+1}-1}{d-1} \Longrightarrow n$ es $O(d^h)$.

Afirmación 44 *Sea* **Q** *la cola de prioridades que utiliza el algoritmo de* DIJKSTRA *para almacenar las etiquetas temporales finitas. Supongamos que* **G** *se almacena como una lista de adyacencia. Si* **G** *es un d-árbol con profundidad h, entonces al aplicar el* DIJKSTRA *sobre* **G** *se tiene que el número de elementos en* **Q** *es a lo más d^h.*

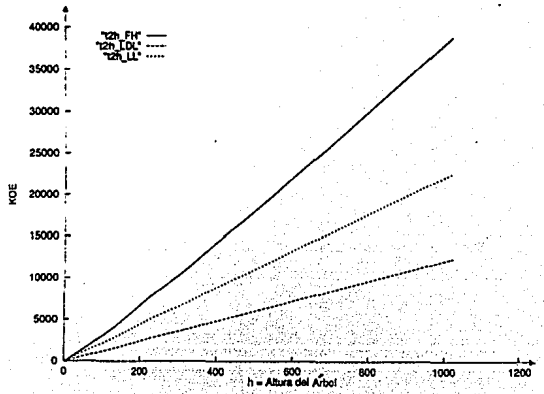


Figura 36: Comparación del DIJSKTRA aplicado a T_{2h}.

Justificación. La demostración se hará por inducción sobre h.

Base de la Inducción. $h = 1$ implica que tenemos una gráfica estrella con d arcos. La Figura 38 muestra un d-árbol con profundidad $h = 1$.

Siguiendo el algoritmo de DIJKSTRA,

$\mathbf{Q}$: $[v_o, 0]$ *borra* v_o $\Longrightarrow \mathbf{Q} = \phi$.

Revisa los vecinos de v_o, son d: $[v_{1,1}, c_{11}], [v_{1,2}, c_{12}], \ldots, [v_{1,d}, c_{1,d}]$ todos con etiqueta infinito; modifica la etiqueta e inserta $v_{1,1}$ ... $v_{1,d}$ en $\mathbf{Q}$.

$\Longrightarrow \mathbf{Q}$: $[v_{1,1}, c_{11}]$-$[v_{1,2}, c_{12}]$- ... -$[v_{1,d}, c_{1,d}]$ $\Longrightarrow |\mathbf{Q}| = d = d^1 = d^h$.

Como cada $v_{1,j}\ \forall\, j = 1, \ldots, d$ son hojas, entonces ya no es posible insertar más datos a $\mathbf{Q}$ en las siguientes iteraciones del algoritmo de DIJKSTRA. Además, para cada una de las siguientes iteraciones del algoritmo el número de elementos en $\mathbf{Q}$ ira decrementandose en uno. Por lo tanto, el tamaño máximo de la Cola de Prioridades es d.

Paso Inductivo. Suponga que la afirmación es válida para $h = k$. Por demostrar que la afirmación es válida para $h = k + 1$. Esto es: si un d-árbol tiene profundidad $(k + 1)$ entonces la Cola de Prioridades tiene a lo más d^{k+1} elementos.

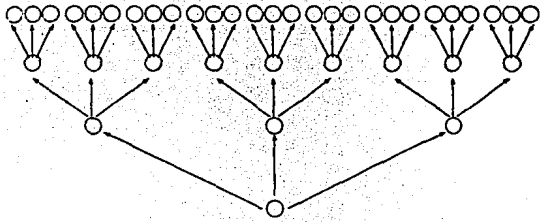


Figura 37: d-árbol con profundidad h, para $d = h = 3$.

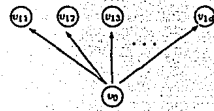


Figura 38: d-árbol con profundidad $h = 1$.

Demostración

Como la afirmación es válida para k se tiene que si la profundidad del árbol es k entonces el tamaño de **Q** es a lo más d^k. Nótese que como el árbol es balanceado, estos d^k elementos de **Q** estan en el nivel k del árbol. Al considerar los elementos del nivel $(k+1)$, supongamos, sin pérdida de generalidad, que los d^k elementos de **Q** tienen d-hijos, cada uno de ellos. Al aplicar un **BorraMin** a la Cola de Prioridades, si el elemento mínimo es vértice del nivel k del árbol, entonces se insertan a lo más sus d hijos en **Q**. Pero esto sucede para cada uno de los d^k elementos del nivel k del árbol, entonces se insertan, en total, a lo más: $d \cdot d^k = d^{k+1}$ elementos.

Nótese que si $d = 1$ el d-árbol es una ruta. De acuerdo con la afirmación 44, el tamaño de la cola de prioridades **Q** es $|\mathbf{Q}| = 1^h = 1$, por lo tanto el desempeño computacional del algoritmo es lineal, como habiamos descrito anteriormente.

Ejemplo 1. Consideremos el d-árbol de la Figura 39. es árbol binario con profundidad $h = 3$. siguiendo el algoritmo de DIJKSTRA.

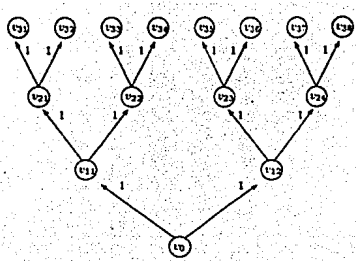


Figura 39: Ejemplo 1: Árbol Binario.

Q: $[v_0, 0]$. El mínimo es v_0 $\Longrightarrow$ **Q** = ϕ.

Revisa los vecinos de v_0, son dos: $[v_{1,1}, 1]$, $[v_{1,2}, 1]$ ambos con etiqueta infinito; modifica etiquetas e inserta $v_{1,1}$ y $v_{1,2}$ en **Q**.

$\Longrightarrow$ **Q:** $[v_{1,1}, 1]$-$[v_{1,2}, 1]$ $\Longrightarrow$ $|\mathbf{Q}| = 2$.

Borra el mínimo, supongamos, sin pérdida de generalidad, que el mínimo es $v_{1,1}$. Revisa los vecinos de $v_{1,1}$, son dos: $[v_{2,1}, 1]$ y $[v_{2,2}, 1]$ con etiqueta infinito, ambos; modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{1,2}, 1]$-$[v_{2,1}, 2]$-$[v_{2,2}, 2]$ $\Longrightarrow$ $|\mathbf{Q}| = 3$.

El mínimo es $v_{1,2}$. Los vecinos de $v_{1,2}$, son dos: $[v_{2,3}, 1]$ y $[v_{2,4}, 1]$ con etiqueta infinito, ambos; modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{2,1}, 2]$-$[v_{2,2}, 2]$-$[v_{2,3}, 2]$-$[v_{2,4}, 2]$ $\Longrightarrow$ $|\mathbf{Q}| = 4$.

Borra el mínimo, supongamos, sin pérdida de generalidad, que el mínimo es $v_{2,1}$ Revisa los vecinos de $v_{1,2}$, son dos: $[v_{3,1}, 1]$ y $[v_{3,2}, 1]$ con etiqueta infinito, ambos; modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{2,2}, 2]$-$[v_{2,3}, 2]$-$[v_{2,4}, 2]$-$[v_{3,1}, 3]$-$[v_{3,2}, 3]$ $\Longrightarrow$ $|\mathbf{Q}| = 5$.

Borra el mínimo, tomemos como mínimo a $v_{2,2}$. Los vecinos de $v_{2,2}$, son dos: $[v_{3,3}, 1]$ y $[v_{3,4}, 1]$ con etiqueta infinito, ambos; modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{2,3}, 2]$-$[v_{2,4}, 2]$-$[v_{3,1}, 3]$-$[v_{3,2}, 3]$-$[v_{3,3}, 3]$-$[v_{3,4}, 3]$ $\Longrightarrow$ $|\mathbf{Q}| = 6$.

Borra el mínimo, tomemos como mínimo a $v_{2,3}$. Revisa los vecinos de $v_{2,3}$, son dos:

$[v_{3,5},1]$ y $[v_{3,6},1]$ con etiqueta infinito, ambos: modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{2,4},2]$-$[v_{3,1},3]$-$[v_{3,2},3]$-$[v_{3,3},3]$-$[v_{3,4},3]$-$[v_{3,5},3]$-$[v_{3,6},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 7$.

El mínimo es $v_{2,4}$. Revisa los vecinos de $v_{2,4}$. son dos: $[v_{3,8},1]$ y $[v_{3,7},1]$ con etiqueta infinito, ambos; modifica etiquetas y los inserta en **Q**.

$\Longrightarrow$ **Q:** $[v_{3,1},3]$-$[v_{3,2},3]$-$[v_{3,3},3]$-$[v_{3,4},3]$-$[v_{3,5},3]$-$[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$
$\Longrightarrow$ $|\mathbf{Q}| = 8$.

Borra el mínimo, tomemos como mínimo a $v_{3,1}$. El vértice $v_{3,1}$ no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,2},3]$-$[v_{3,3},3]$-$[v_{3,4},3]$-$[v_{3,5},3]$-$[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 7$.

Borra el mínimo, tomemos como mínimo a $v_{3,2}$. El vértice $v_{3,2}$ no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,3},3]$-$[v_{3,4},3]$-$[v_{3,5},3]$-$[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 6$.

Borra el mínimo, tomemos como mínimo a $v_{3,3}$. El vértice $v_{3,3}$ no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,4},3]$-$[v_{3,5},3]$-$[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 5$.

Borra el mínimo, tomemos como mínimo a $v_{3,4}$. El vértice $v_{3,4}$ no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,5},3]$-$[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 4$.

Borra el mínimo, tomemos como mínimo a $v_{3,5}$. El vértice $v_{3,5}$ no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,6},3]$-$[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 3$.

Borra el mínimo, tomemos como mínimo a $v_{3,6}$, el cual no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{3,7},3]$-$[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 2$.

Borra el mínimo, tomemos como mínimo a $v_{3,7}$. Este vértice no tiene vecinos. $\Longrightarrow$ **Q:** $[v_{3,8},3]$ $\Longrightarrow$ $|\mathbf{Q}| = 1$.

El mínimo es $v_{3,8}$ y no tiene vecinos $\Longrightarrow$ **Q:** $\emptyset$ $\Longrightarrow$ $|\mathbf{Q}| = 0$.

Como podemos observar en este ejemplo, el tamaño máximo de la cola de prioridades es $||\mathbf{Q}|| = 8 = 2^3 = 2^h$.

Ejemplo 2. Consideremos el d-árbol de la Figura 40, es árbol binario con profundidad $h = 3$. Siguiendo el algoritmo de DIJKSTRA,

Q: $[v_0,0]$ *borra* el mínimo $\Longrightarrow \mathbf{Q} = \phi$.
Revisa los vecinos de v_0, son 2: $[v_{1,1},1]$, $[v_{1,2},8]$ ambos con etiqueta infinito; modifica etiquetas e inserta $v_{1,1}$ y $v_{1,2}$ en **Q**.

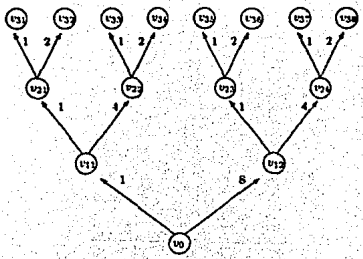


Figura ¡0: Ejemplo 2: Árbol Binario.

$\Longrightarrow$ **Q:** $[v_{1,1},1]$-$[v_{1,2},8]$ $\Longrightarrow$ $\|Q\| = 2$.

El mínimo es $v_{1,1}$ Revisa los vecinos de $v_{1,1}$, son dos: $[v_{2,1},1]$ y $[v_{2,2},4]$ con etiqueta infinito, ambos; modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{2,1},2]$-$[v_{2,2},5]$ $\Longrightarrow$ $\|Q\| = 3$.

El mínimo es $v_{2,1}$ Revisa los vecinos de $v_{2,1}$, son dos: $[v_{3,1},1]$ y $[v_{3,2},3]$ con etiqueta infinito, ambos; modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{2,2},5]$-$[v_{3,1},3]$-$[v_{3,2},4]$ $\Longrightarrow$ $\|Q\| = 4$.

El mínimo es $v_{3,1}$ Revisa los vecinos de $v_{3,1}$, no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{2,2},5]$-$[v_{3,2},4]$ $\Longrightarrow$ $\|Q\| = 3$.

El mínimo es $v_{3,2}$ Revisa los vecinos de $v_{3,2}$, no hay vecinos.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{2,2},5]$ $\Longrightarrow$ $\|Q\| = 2$.

El mínimo es $v_{2,2}$ Revisa los vecinos de $v_{2,2}$, son dos: $[v_{3,3},1]$ y $[v_{3,4},2]$ con etiqueta infinito, ambos; modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{3,3},6]$-$[v_{3,4},7]$ $\Longrightarrow$ $\|Q\| = 3$.

El mínimo es $v_{3,3}$ Revisa los vecinos de $v_{3,3}$, no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$-$[v_{3,4},7]$ $\Longrightarrow$ $\|Q\| = 2$.

El mínimo es $v_{3,4}$ Revisa los vecinos de $v_{3,1}$: no tiene vecinos.

$\Longrightarrow$ **Q:** $[v_{1,2},8]$ $\Longrightarrow$ $\|Q\| = 1$.

El mínimo es $v_{1,2}$ Revisa los vecinos de $v_{1,4}$, son dos: $[v_{2,3}, 1]$ y $[v_{2,4}, 4]$ con etiqueta infinito, ambos: modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q**: $[v_{2,3}, 9]$-$[v_{2,4}, 12]$ $\Longrightarrow$ $||\mathbf{Q}|| = 2$.

El mínimo es $v_{2,3}$ Revisa los vecinos de $v_{2,3}$, son dos: $[v_{3,5}, 1]$ y $[v_{3,6}, 2]$ con etiqueta infinito, ambos: modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q**: $[v_{2,4}, 12]$-$[v_{3,5}, 10]$-$[v_{3,6}, 11]$ $\Longrightarrow$ $||\mathbf{Q}|| = 3$.

El mínimo es $v_{3,5}$ Revisa los vecinos de $v_{3,5}$: no tiene vecinos.

$\Longrightarrow$ **Q**: $[v_{2,4}, 12]$-$[v_{3,6}, 11]$ $\Longrightarrow$ $||\mathbf{Q}|| = 2$.

El mínimo es $v_{3,6}$ Revisa los vecinos de $v_{3,6}$: no tiene vecinos.

$\Longrightarrow$ **Q**: $[v_{2,4}, 12]$ $\Longrightarrow$ $||\mathbf{Q}|| = 1$.

El mínimo es $v_{2,4}$ Revisa los vecinos de $v_{2,4}$, son dos: $[v_{3,7}, 1]$ y $[v_{3,8}, 2]$ con etiqueta infinito, ambos: modifica etiquetas e inserta los vértices en **Q**.

$\Longrightarrow$ **Q**: $[v_{3,7}, 13]$-$[v_{3,8}, 14]$ $\Longrightarrow$ $||\mathbf{Q}|| = 2$.

El mínimo es $v_{3,7}$ Revisa los vecinos de $v_{3,7}$: no tiene vecinos.

$\Longrightarrow$ **Q**: $[v_{3,8}, 14]$ $\Longrightarrow$ $||\mathbf{Q}|| = 1$.

El mínimo es $v_{3,8}$ y no tiene vecinos, entonces Q: $\emptyset$ $\Longrightarrow$ $||\mathbf{Q}|| = 0$.

Para este ejemplo tenemos que el tamaño máximo de **Q** es: $||\mathbf{Q}|| = 4 \leq 2^3$.

Ahora, realizaremos análisis sobre el comportamiento de las implantaciones concretas del algoritmo DIJKSTRA aplicado los d-árboles.

Listas Simplemente Ligadas.

Analizaremos esta implantación concreta en términos de operaciones elementales. Tenemos que las operaciones **Vacio**, **Principio** e **CreaPQ** gastan una operación elemental, cada **inserta** gasta dos y cada **BorraMin** requiere $(i+4)$ operaciones elementales, pero por la Afirmación 44, i es a lo más d^k, por lo tanto:

$$\begin{aligned}\textbf{Koe\_PQ} &= (n+1)\,\textbf{BorraMin} + (n+1)\,\textbf{Inserta} + (n+1)\,\textbf{Vacio} \\ &\quad + 1\cdot\textbf{Principio} + 1\cdot\textbf{CreaPQ}.\end{aligned}$$

$$\textbf{Koe\_PQ} \leq (n+1)\cdot d^k + 3\cdot(n+1) + 2 \leq (n+1)\cdot(d^k+3) + 2.$$

Por lo tanto, tenemos que el número de operaciones elementales, para esta implantación es lineal en n, pero sustituyendo el valor de n tenemos:

$$\text{Koe\_PQ} \leq (d^h + 3) \cdot \left(\frac{d^{h+1} - 1}{d - 1} + 1 \right) + 2 = O(d^{2h}).$$

Así que, el número de operaciones elementales para esta implantación es polinomial en d, número de hijos en cada nodo. Pero d^h es el número máximo de elementos que existe en el último nivel del d-árbol de profundidad h, así que podemos decir que el algoritmo es cuadrático en el número máximo de hojas de un d-árbol.

Listas Doblemente Ligadas.

De acuerdo a la demostración del Teorema 13: $\sum_{i=1}^{n} t(Q.Inserta, i) \leq \sum_{i=1}^{n} d_i$, donde d_i es distancia entre la posición correcta de i, el nuevo elemento a insertar, y ui, la posición del último elemento insertado, pero esta distancia es a lo más d^h, así que:

$$\sum_{i=1}^{n} t(Q.Inserta.i) \text{ es } O(n \cdot d^h) \quad \text{pero } n \text{ es } O(d^h)$$

$$\Longrightarrow \sum_{i=1}^{n} t(Q.Inserta.i) \text{ es } O(d^{2h}).$$

Pero como el algoritmo DIJKSTRA tiene desempeño computacional, en el peor de los casos, de:

$$T_E(n) = \Theta(n) + \sum_{i=1}^{n} t(Q.Inserta, i) \Longrightarrow T_E(n) = \Theta(n) + O(d^{2h}).$$

Finalmente, tenemos que el desempeño computacional del DIJKSTRA resulta ser una función lineal en n, número total de vértices, y polinomial en d, número de hijos por vértice.

Fibonacci Heaps.

Sabemos que el Desempeño Computacional del Algoritmo DIJSKTRA, para esta implantación concreta de F-Heaps, depende específicamente de la operación **BorraMin** y ya habíamos demostrado que ésta requiere tiempo $\Theta(\log_2 i)$, donde i es el tamaño de la cola. Por otra parte, sabemos que el tamaño de la cola es a lo más d^h por lo que: $i \leq d^h$, entonces:

$\log_2 i \leq \log_2 d^h \leq h \cdot \log_2 d$, por lo tanto: $\sum_{i=1}^{n} t(Q.Inserta, i) = \Theta(n \cdot h \log_2 d)$.

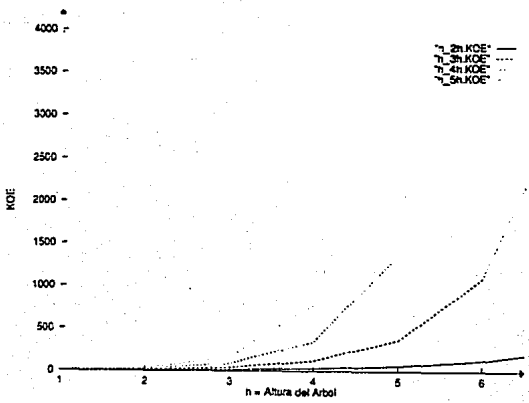


Figura 41: Desempeño Computacional del DIJKSTRA aplicado a d-árbol, usando F-Heaps.

Finalmente, tenemos que el Desempeño Computacional del algoritmo para esta implantación de F-Heaps, en el peor de los casos, es $T_{\mathcal{E}}(n) = \Theta(n) + \Theta(n \cdot h \log_2 d)$. Esta función es lineal en n, número total de nodos, y polinomial en d, número de hijos por vértice.

La Figura 41 muestra el comportamiento del algoritmo de DIJKSTRA, utilizando F-Heaps, aplicado a gráficas cuya estructura representa un d-árbol. Para esta gráfica, el eje **X** representa a h, la altura del árbol y el eje **Y** indica el número de operaciones elementales realizadas por el algoritmo.

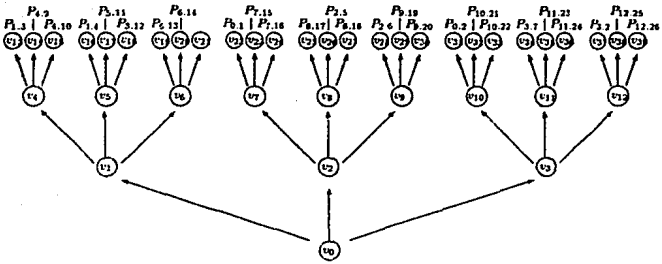


Figura 42: Coloración de caminos en un árbol.

4 Coloración de Caminos

Regla para Colorear Caminos.

> Sea la gráfica G un árbol. Supongamos que tenemos todos los colores disponibles. Si un vértice v tiene k-hijos, se pintan las aristas de $(k-1)$ caminos desde v hasta un vértice terminal *hoja*. Cada camino de diferente color. El k-ésimo camino de un vértice v tiene el color de algún camino que fue coloreado con anterioridad, esto es, tiene el color de un camino que pertenezca a un vértice antecesor a v. No se permite que una arista tenga más de un color.

Nótese que si el número de hojas (vértices terminales) en un árbol es H, entonces el número de colores utilizados para iluminar los caminos es a lo más H.

La Figura 42 nos muestra un ejemplo de este tipo de coloración. La notación $P_{i,j}$ representa la ruta que inicia en el vértice i y tiene el color j. Por ejemplo, la ruta $P_{0,1}$ es la ruta que inicia en el vértice v_o y tiene el color 1, en particular, esta ruta esta formada por los vértices: v_0, v_2, v_7, v_{22}.

Definición.

> Diremos que ω es un **descendiente en ruta** de v si ω está en la ruta del mismo color que v, esto es: ω tiene el mismo color que v.

Ejemplo 1. Regresando a la Figura 42 tenemos que

+ v_{16} es descendiente en ruta de v_1, pero no lo es de v_o.
+ v_o tiene dos rutas coloreadas.
  Los descendientes en una ruta de v_o son: v_{22}, v_7 y v_2
  Los descendientes en otra ruta de v_o son: v_{31}, v_{10} y v_3

Ejemplo 2. Considerese una gráfica T_{kh}, observamos que:

+ hay exactamente k rutas;
+ todos los vértices ω con $\omega \neq v_o$ son descendientes en ruta de v_o.

Teorema 45 .

Sea $\delta^-(v)$ *el grado exterior de* v, *el número de aristas que 'salen' de* v. *Sea* $d(s,v)$ *el costo de la ruta que va de* s *a* v.

Si $||Q||_u$ *es el tamaño de la cola* Q *al ingresar* u *en ella, entonces*

$$||Q||_u = \sum_{v \in \mathcal{V}_u} ||W_v||$$

donde $\mathcal{V}_u = \{ v \,/\, \delta^-(v) \geq 2 \;\; y \;\; d(s,v) < d(s,u) \}$

$y \;\; W_v = \{ \omega \,/\, \omega$ *es descendiente en ruta de* $v \;\; y \;\; d(s,\omega) > d(s,v) \}$

$||Q||_u$ representa el número de vértices con etiqueta temporal que fueron elegidos antes que u.

Analicemos ahora este resultado para la implantación concreta de Fibonacci-Heaps hemos venido manejando en este trabajo.

Nótese que el número operaciones que se aplican sobre Q es de orden logarítmico, entonces

$$\sum_{k=1}^{n} \log ||Q||_{u_k} = \log \left[\prod_{k=1}^{n} ||Q||_{u_k} \right] \quad \ldots \star$$

la suma se está aplicando sobre el orden topolgico en que el DIJKSTRA elige los vértices.

Si Q tiene tamaño constante, como en las gráficas $\mathcal{L}_n$ y $T_{kh}, h >> k$, tenemos que

$$\star = \log \prod_{k=1}^{n} C = \log C^n = n \log C$$

con lo que reafirmamos que el algoritmo de DIJKSTRA tiene desempeño computacional de orden lineal.

Si hay una una u tal que $||Q||_u = \Omega(n)$, entonces

$$||Q||_{u+i} \geq ||Q||_u - 1, \qquad \text{para } i = 1, \ldots, (n-u)$$

$$\geq \Omega(\log n^n) = \Omega(n \cdot \log n)$$

FALTA PAGINA

No.___a la___

95

VII Conclusiones

Durante la investigación de este trabajo hemos encontrado que el Desempeño Computacional del Algoritmo de DIJKSTRA depende en gran medida de la estructura de la red y de la forma como se representen las etiquetas temporales en la implantación del algoritmo.

Hemos notado que la dificultad de la red depende del grado externo (out degree) de los vértices, del número de ciclos existentes en la red y del grado de desorden con que se le presenten los costos de las aristas al algoritmo.

De esta forma, tenemos evidencias concretas sobre la adaptabilidad del Algoritmo de DIJKSTRA, incluso hasta encontrar *Adaptabilidad Óptima*. Estas versiones adaptivas del algoritmo se logran mediante estructuras de datos sofisticadas para la implantación de la cola de prioridades, la cual representa las etiquetas temporales en el algoritmo.

Las versiones adaptivas del Algoritmo de DIJKSTRA consumen menos recursos de cómputo, en particular su Desempeño Computacional es mejor que en el peor de los casos y en el caso promedio. Por ello, aplicar estas variables adaptivas en algoritmos de Optimización Combinatoria y Flujo en Redes, que utilizan el Problema de la Ruta Más Corta como una operación, mejorará el desempeño computacional de los mismos.

Consideramos que el Análisis y, en especial, la Adaptabilidad de Algoritmos en el área de Optimización Combinatoria y Teoría de Redes es un campo muy amplio por explorar. Explorarlo nos llevará a descubrir, modificar y diseñar algoritmos que no sólo mejoren su desempeño computacional, sino que además sean capaces de aprovechar la estructura particular de los datos de entrada.

Los Análisis de Adaptabilidad para el Problema de la Ruta Más Corta, descritos en este trabajo, son los primeros en efectuarse en esta área y son apenas una evidencia de lo que hay por realizar en el campo de la Teoría de Redes.

El Análisis de Adaptabilidad para el Problema de la Ruta Más Corta, como hemos visto en el presente trabajo, requiere establecer medidas que cuantifiquen la dificultad de la red; identificar entre los algoritmos que resuelven la Ruta Más Corta cuales son adaptivos y cuales no; revisar entre los algoritmos que no son adaptivos cuales pueden llegar a serlo medi-

ante modificaciones: diseñar algoritmos adaptivos y establecer propiedades que garanticen máxima adaptabilidad.

Estamos concientes de que aún falta formalizar *medidas de complejidad* de la red y que nuestros análisis de adaptabilidad únicamente están enfocados en el Algoritmo de DIJKSTRA, el cual resuelve el problema de la Ruta Más Corta de un vértice a cada uno de los otros vértices en una red con costos positivos, pero existen más algoritmos por investigar, por ejemplo algoritmos que permitan arcos con longitudes negativas o algoritmos que resuelvan la Ruta Más Corta entre todo par de vértices.

Debido a que el problema de Ruta Más Corta es básico para el problemas de Teoría de Redes, resulta interesante saber, con exactitud, cual es el comportamiento de los Algortimos de Teoría de Redes que utilizan Ruta Más Corta cuando emplean versiones adaptivas. Por otra parte, debemos ampliar más el área de investigación hacia todo tipo de Algoritmos, no sólo en Teoría de Redes u Optimización Combinatoria, sino en todos los campos de aplicación.

Como trabajo futuro, también consideramos interesante estudiar la adaptabilidad de algoritmos en el modelo de cómputo en paralelo.

VIII Apéndices

A Algoritmos para la Ruta Más Corta

1 Ruta Más Corta de s a t

Los algoritmos que resuelven la RMC entre dos nodos específicos, $s \rightarrow t$ se auxilian del método que busca la RMC del origen a cada uno de los otros nodos en **G**. En esta sección presentamos algunos de los métodos más tradicionales para resolver éste problema.

1.1 Ecuaciones de Bellman.

Las **Ecuaciones de Bellman** representan una solución analítica para resolver el problema de encontrar la RMC entre dos nodos específicos.

Se enumeran los vértices de 1 a n, con $s = 1$ y $d = n$.

Sea u_j = Longitud de la RMC del nodo 1 al nodo j.

Sea a_{ij} = Longitud del arco (i,j):

$$a_{ij} = \begin{cases} a_{ij} & \text{si } \exists (i,j) \in A \\ \infty & \text{e.o.c.} \end{cases}$$

Las **Ecuaciones de Bellman** indican que la longitud de la RMC satisface

$$(I): \qquad u_j = \begin{cases} 0 & j = 1 \\ \min_{k \neq j} \{u_k + a_{kj}\} & j = 2, 3, \ldots, n. \end{cases}$$

Estas ecuaciones se cumplen siempre que la red *no contenga ciclos negativos*. Nótese que la longitud de la RMC está bien definida, pues cada a_{kj} es un número finito, por lo cual si la solución existe es única. Al resolver (I) se obtiene un árbol dirigido y enraizado, con raíz s, tal que la longitud de la única ruta del nodo s al j es u_j, si existe, a tal se le conoce como Árbol de RMC.

1.2 Redes Acíclicas.

Las **Ecuaciones de Bellman** resultan más faciles de resolver si se tiene una *red acíclica*, ya que los arcos pueden ser numerados. Lo cual está fundamentado en el siguiente teorema de la *Teoría de Gráficas*:

Teorema. Una gráfica dirigida es *acíclica* si y sólo si sus nodos pueden ser *numerados* de tal forma que para cada arco (i,j) se tiene que $i < j$.

La prueba constructiva de este teorema muestra que al numerar G se obtiene un algoritmo cuya complejidad es $O(n^2)$.

Al asumir que los nodos han sido numerados las *Ecuaciones de Bellman* se transforman en:

$$(AN): \qquad u_j = \begin{cases} 0 & j = 1 \\ \min_{k<j}\{u_k + a_{kj}\} & j = 2,3,\ldots,n \end{cases}$$

Resolver las ecuaciones (AN) requiere de una simple sustitución, pues:

$$\begin{cases} u_1 = 0 & \\ u_2 = f(u_1) & \text{depende de } u_1 \\ u_3 = f(u_1, u_2) & \text{depende de } u_1, u_2 \\ \quad\ddots & \quad\ddots \\ u_{k+1} = f(u_1, u_2, \ldots, u_k) & \text{depende de } u_1, \ldots, u_k \end{cases}$$

o bien:

$$\begin{cases} u_2 = min\{u_i + a_{i2}\} & \forall\,(i,2) \in A \\ u_3 = min\{u_i + a_{i3}\} & \forall\,(i,3) \in A \\ \quad\ddots & \quad\ddots \\ u_{k+1} = min\{u_i + a_{ik}\} & \forall\,(i,k) \in A \end{cases}$$

Solucionar las n ecuaciones anteriores requiere:

$0+1+2+3+\ldots+(n-1) = n(n-1)/2$ sumas y

$0+0+1+2+\ldots+(n-2) = (n-1)(n-2)/2$ comparaciones

Por lo que, resolver el problema de la RMC en una red acíclica tiene desempeño computacional de $O(n^2)$. Como la red es acíclica, resulta obvio que *no* contiene *ciclos negativos*, entonces no hay que preocuparse por las longitudes de los arcos, las cuales pueden ser negativas. Los cálculos se realizan, en este caso, sin complicación alguna.

1.3 Algoritmo de Dijkstra.

El Algoritmo de Dijkstra es uno de los *más eficientes* para resolver el problema de la Ruta Más Corta, en una red G con longitudes de arco *no negativas*. En el Capítulo II de este trabajo se desarrolla detalladamente.

1.4 Método de Bellman-Ford.

Una generalización del algoritmo de DIJKSTRA para gráficas donde las longitudes de arco pueden ser negativas, pero sin contener circuitos negativos, fue hecha por Ford y es denominada **Método de Bellman-Ford**, consiste en resolver las ecuaciones de Bellman por aproximaciones sucesivas.

Se define:

- $u_j^{(k)}$ = longitud de una ruta más corta del origen s a j de tal forma que la ruta contiene a lo más k arcos.
- $u_j^{(k)} = \begin{cases} 0 & \text{si } j = 1 \\ a_{ij} & \text{e.o.c.} \end{cases}$

El cálculo de las primeras $(m+1)$ aproximaciones de orden m es:

$$u_j^{(m+1)} = \min \{ u_j^{(m)}, \min_{k \neq j} \{ u_k^{(m)} + a_{kj} \} \} \quad \ldots (BF)$$

se observa que para cada nodo j: $u_j^{(1)} \geq u_j^{(2)} \geq \ldots \geq u_j^{(k)} \geq \ldots$

Si la red no contiene circuitos negativos, entonces *existe* la RMC del origen s a cada nodo j, sin repetir nodos. Si la red tiene n nodos entonces *existe* una ruta con no más de $(n-1)$ arcos por tanto: $u_j = u_j^{(n-1)} \ \forall j \in A$. Este algoritmo tiene un desempeño computacional de $O(n^3)$.

1.5 Modificaciones de Yen.

Como un mejoramiento en eficiencia a las ecuaciones de Bellman-Ford son consideradas las **Modificaciones de Yen**. Las aproximaciones u_j^k, de Bellman-Ford, no hacen uso de la mejor información disponible en cada iteración. Por ejemplo, se calcula $u_j^{m+1} = f(u_1^m, u_2^m, \ldots, u_n^m)$ aunque las u_k^{m+1} k = 1, 2 ... (j-1), ya hayan sido evaluadas.

Se define:

- (i,j) va *hacia adelante* si $i < j$.
- (i,j) va *hacia atrás* si $i > j$.
- Una ruta tiene un *cambio de dirección* cuando un arco hacia atrás es seguido de uno hacia adelante o viceversa.

▷ $u_j^{(k)}$ = longitud de la RMC de s a j de tal forma que hay a lo más $(k-1)$ cambios de dirección.

▷ $u_j^{(0)} = \begin{cases} 0 & \text{si } j = 1. \\ a_{ij} & \text{si } (i,j) \in A. \\ \infty & \text{e.o.c.} \end{cases}$

Entonces las Ecuaciones de Bellman se transforman en:

$$(Y): \begin{cases} u_j^{(k+1)} = \min\{ u_j^{(k)}, \min_{i<j}\{ u_j^{(k+1)} + a_{ij} \} \} & \forall\, k \text{ par.} \\ u_j^{(k+1)} = \min\{ u_j^{(k)}, \min_{i>j}\{ u_j^{(k+1)} + a_{ij} \} \} & \forall\, k \text{ impar.} \\ u_j = u_j^{(n-1)}. \end{cases}$$

Cada una de las ecuaciones en (Y) se resuelve sobre $n/2$ opciones, en vez de n. Ahora se realizan a lo más $n^3/2$ sumas y $n^3/2$ comparaciones, por tanto este método reduce el desempeño computacional del algoritmo en una razón de dos. Una ventaja extra es que el almacenamiento puede reducirse en un orden de dos, pues una vez que $u_j^{(m+1)}$ es calculada, ésta reemplaza a $u_j^{(m)}$.

2 Ruta Más Corta entre todo par de nodos.

Para este problema, se desea calcular la RMC de cada nodo v de la gráfica a cada uno de los otros $(n-1)$ nodos, teniendose en total $n \cdot (n-1)$ rutas.

2.1 Multiplicación Matricial

Se define:

u_{ij} = Longitud de la RMC de i a j.

$u_{ij}^{(m)}$ = Longitud de la RMC de i a j, tal que la ruta no tiene más de m arcos.

a_{ij} = Longitud del arco (i,j).

Si $a_{ii} = 0 \ \forall\ i$.

$$(MM): \begin{cases} u_{ii}^{(0)} = 0 & \forall\ i. \\ u_{ij}^{(0)} = \infty & \forall\ i \neq j. \\ u_{ij}^{(m+1)} = \min\{ u_{ik}^{(m)} + a_{kj} \} & \forall\ k. \\ u_{ij} = u_{ij}^{(n-1)} & \text{ruta final.} \end{cases}$$

El desempeño computacional del algoritmo se espera que sea a lo más $O(n^4)$ al aplicar n veces el algoritmo de Bellman-Ford.

Se difine la operación: $P = [p_{ij}] = A \odot B$. con $p_{ij} = \min\{a_{ik} + b_{kj}\}\ \forall k$.

Sean:

$U^m = [u_{ij}^{(m)}]$ = matriz de aproximaciones de orden m. y

$A = [a_{ij}]$ = la matriz de longitudes de arco.

entonces:

$$\begin{cases} U^{(1)} &= U^{(0)} \otimes A \\ U^{(2)} &= U^{(1)} \otimes A = (U^{(0)} \otimes A) \otimes A) \\ \ddots &= \ddots \\ U^{(n-1)} &= U^{(n-2)} \otimes A \\ &= ((\ldots (U^{(0)} \otimes A) \ldots \otimes A). \end{cases}$$

Observamos que la *multiplicación* es asociativa, entonces tenemos:

$$\begin{aligned} U^{(n-1)} &= U^{(n-2)} \otimes A \\ &= ((\ldots (U^{(0)} \otimes A) \ldots \odot A) \\ &= (U^{(0)} \otimes A) \otimes A \ldots \otimes A \\ &= (A) \otimes A \odot \ldots \otimes A = A^{(n-1)}. \end{aligned}$$

Ahora, como $U^{(0)}$ es la matriz identidad: $U^{(0)} \otimes A = A$

Este método requiere $\log_2 n$ *multiplicaciones* de matrices cada una de las cuales tiene complejidad $O(n^3)$, por tanto se tiene que el desempeño computacional resulta ser $O(n^3 \log_2 n)$. Es fácil ver que el Algoritmo de BELLMAN-FORD puede ser implantado utilizando este método matricial.

Sea: $u^{(m)} = (u_1^{(m)}, u_2^{(m)}, \ldots, u_n^{(m)})$ y $u^{(0)} = (0, \infty, \infty, \ldots, \infty)$,

entonces $u^{(m+1)} = u^{(m)} \otimes A = u^{(0)} \otimes A^m$.

2.2 Método de Floy-Warshal

$u_{ij}^{(m)}$ = Longitud de la RMC de i a j de tal forma que la ruta no pase por los nodos $m, m+1, \ldots, n$, excepto i y j.

En una ruta del nodo i al j que no use los nodos $m+1, m+2, \ldots, n$ sucede que:

1.- No pasa por el nodo $m \quad \Longrightarrow u_{ij}^{(m+1)} = u_{ij}^{(m)}$.

2.- Si pasa por el nodo $m \quad \Longrightarrow u_{ij}^{(m+1)} = u_{im}^{(m)} + u_{mj}^{(m)}$.

Entonces

$$(FW): \begin{cases} u_{ij}^{(1)} & = a_{ij}. \\ u_{ij}^{(m+1)} & = \min\{\, u_{ij}^{(m)},\ u_{im}^{(m)} + u_{mj}^{(m)} \,\}. \\ u_{ij} & = u_{ij}^{(n-1)}. \end{cases}$$

Se observa que en las igualdades (FW) hay exactamente $n(n-1)(n-2)$ ecuaciones y cada una de ellas requiere una suma y una comparación para ser resueltas ($i, j, m = 1, 2, \ldots n ; i \neq j ; i \neq m ; j \neq m$). Entonces, el algoritmo **Floy-Warshal**, tiene el mismo desempeño computacional del algoritmo de **Bellman-Ford**.

Si las longitudes de los arcos son positivas y se aplica el algoritmo de Dikjstra n veces para n orígenes, se observa que se requieren el mismo número de sumas y comparaciones que el método original.

2.3 Método de Spira

Spira [42] propone un algoritmo para resolver el problema de la RMC entre cualquier par de vértices. En el peor de los casos el desempeño computacional está acotado por $O(n^3 \log n)$, sin embargo, Spira muestra que el número esperado de operaciones es de $O(n^2(\log n)^2)$.

Fredman [30] en 1976 publica que el número de comparaciones puede ser reducido a $O(n^2 \log n)$.

2.4 Método de Fredman

En 1976, Fredman [30] muestra que para n fija el problema, con arcos no negativos, puede ser resuelto en un tiempo $O(n^{2.5})$.

Su procedimiento utiliza una *tabla pre-compilada* la cual es usada para todos los problemas de un tamaño n dado, pero el mejor método para compilar la tabla requiere un tiempo $O(n^3 \log(\log n)/\log n)$. Dado el desempeño computacional del algoritmo y su prueba no muy convincente, este resultado es más teórico que práctico.

2.5 RMC en una Red con Arcos No Dirigidos

Sea **G** una gráfica no dirigida. Se consideran dos casos:

1. **Las Longitudes del arco son no negativas.**
   Para cada arco $\{ i, j \}$ sin dirección se generan los arcos: (i,j) y (j,i) con $a_{ij} = a_{ji}$. En este caso se resuelve normalmente con cualquier implantación del algoritmo de DIJSKTRA.
2. **Las Longitudes del arco negativas.**
   ▷ No se pueden generar dos arcos dirigidos ya que formarían un circuito negativo.
   ▷ Es posible plantear este problema como un *Acoplamiento no-bipartito con pesos.* Los arcos de longitud no negativa son permitidos, pero no así los circuitos negativos.

3 Algoritmos de Descomposición

Dado que los Algoritmos de RMC entre todo par de vértices en una red resultan difíciles de implantar en una computadora electrónica cuando **G** resulta ser muy grande y densa, se proponen algoritmos de **Descomposición** que facilita el análisis de la red.

3.1 Algoritmo de G. Mills

G. Mills [29] propone un algoritmo de DESCOMPOSICIÓN para el problema de la RMC que facilita el análisis de tal red y, además, optimiza el almacenamiento de la misma.

Definiciones y Notación.

- ▷ S es el conjunto de todos los nodos en X y Y.
- ▷ El conjunto de arcos, *red*, asociado con S está formado por todos los arcos cuyos nodos pertenecen a S.
- ▷ T como el conjunto de los nodos en X y Z.
- ▷ El conjunto de arcos, *red*, asociado con T se forma por todos los arcos cuyos nodos están en T.
- ▷ Sea V la *red coneza*, que posee todos los arcos asociado con S.
- ▷ Sea W la *red coneza*, que contiene todos los arcos asociado con T.
- ▷ Sea U la *intersección*, de V y W.

▷ **Barrera:**

Conjunto X de nodos que divide a **N** en dos conjuntos, Y y Z, de tal forma que resulta imposible ir de un nodo $y \in Y$ a un $z \in Z$ (o viceversa) sin pasar al menos por un nodo $x \in X$. Es decir, no es posible construir un camino de cualquier $y \in Y$ a cualquier $z \in Z$ que no contenga nodos de X.

Panorámica intuitiva del algoritmo

La Barrera X divide la red en dos partes: S y T, se construyen las redes V y W. Se aplica por separado un método de RMC, auxiliado por matrices esperando obtener la RMC entre cualquier par de nodos en S y en T. Por definición de S y en T se tiene que los $x \in X$ son nodos en común para ambas partes, por lo tanto se puede construir una ruta de algún $y \in Y$ a un $z \in Z$ que utilice al menos un $x \in X$, resta encontrar al nodo x que minimiza la suma para las dos longitudes obtenidas antes.

3.2 Algoritmo de T.C. Hu

T.C. Hu [14] asegura que si: $m < n \cdot (n-1)$, esto es, si **G** tiene menos de $n \cdot (n-1)$ arcos, entonces es posible *descomponer* la red y obtener entre cada parte la RMC entre todo par de nodos.

Este algoritmo de descomposición *disminuye* tanto la cantidad de operaciones como los requerimientos de almacenamiento de una computadora.

B Tipo de Datos Concretos

En este anexo describiremos detalladamente los *TDC Listas Ligadas, Colas Binomiales, Fibonacci Heaps y Radix Heaps*, para el *TDA* Colas de Prioridades.

Considerando que estamos usando la notación de *POO*, llamaremos a las implantaciones concretas, una vez especificadas sus operaciones, *TDC* o *Clase*. La implantación concreta de estos tipos de datos ha sido hecha en el *LOO* EIFFEL [24, 25].

1 Listas Ligadas

Implantaciones concretas de las *Listas Ligadas* son ampliamente conocidas en la literatura [9, 12, 16, 21, 15, 24]. Para el presente trabajo describiremos como implantar una cola de prioridades ulizando *listas simplemente ligadas*, y *listas doblemente ligadas* ambas auxiliadas con un apuntador de acceso directo al elemento con menor prioridad en la cola. También, describiremos una implantación de una cola de prioridades con *Listas Ligadas* para la cual se tenga acceso directo a cualquier elemento en la cola. Denominaremos **Q** a la cola de prioridades.

1.1 Listas Simplemente Ligadas

Para esta implantación se mantiene un apuntador, llamado $MinH$, al elemento con mínima prioridad en la cola **Q**. Llamemos a esta implantación concreta *Clase pq_lsl*.

Descripción de las operaciones.

Inserta.- Añade un nuevo elemento en cualquier lugar de la lista. Digamos que lo agrega en la última posición, de esta forma los datos quedan almacenando de acuerdo a la forma en que fueron llegando. Requiere tiempo de ejecución $\Theta(1)$.

EncuentraMin.- Como la cola **Q** tiene un apuntador al nodo mínimo, esta operación sólo indica el nodo al cual está apuntando. El tiempo de ejecución es de $\Theta(1)$.

BorraMin.- Guarda en una variable auxiliar el nodo mínimo para luego regresarlo, quitarlo de la cola y buscar en *toda* la Cola al nuevo elemento mínimo. Requiere tiempo de ejecución $O(n)$.

Decrementa_Llave.- Busca, en toda la lista, el vértice al que se le hará el cambio, lo decrementa y verifica si el elemento mínimo se modifica, en tal caso actualiza el apuntador del mínimo. Requiere tiempo de ejecución $O(n)$.

1.2 Listas Doblemente Ligadas

Esta implantación mantiene ordenados a los elementos de la lista según su prioridad en la cola **Q**. Denominamos a esta implantación concreta *Clase pq_ldl*.

Descripción de las operaciones

Inserta(x **: dato).-** Debe buscar en **Q** el lugar que le corresponde al dato de acuerdo a su llave. El proceso puede resumirse de la siguiente manera:

Sea ui un apuntador al último elemento insertado.

Si $llave(ui) \leq llave(x)$ **Entonces**

busca a la derecha de ui la posición correcta de x e insertalo;

En otro caso

busca a la izquierda de ui el lugar correcto de x e insertalo.

El tiempo de ejecución de esta operación depende del número de elementos que hay entre ui y la posición correcta de x. El desplazamiento de la posición de la operación anterior a la posición actual es d_x, la Figura 43 ilustra este hecho. Por tanto, **Inserta** gasta tiempo $O(d_x)$.

EncuentraMin.- Regresa el primer elemento de la Cola. Requiere tiempo $\Theta(1)$.

BorraMin.- Regresa y elimina al primer elemento de la cola. Gasta tiempo $\Theta(1)$.

Decrementa_LLave.- Debe buscar el vértice para el cual se hará el cambio, decrementarlo y verificar si **Q** queda en orden, si no es así debe reubicar el dato que ha quedado en desorden. Requiere tiempo $O(n)$.

Un ejemplo de la operación **Inserta** para $\ell = \{5, 4, 3, 2, 1\}$ se presenta en la Figura 44.

1.3 Listas Ligadas con apuntadores externos

En las versiones anteriores de los *TDC* para la cola de prioridades, la operación **Decrementa_Llave**, requiere tiempo de ejecución de $O(n)$, ya que buscan sobre toda la cola al vértice a modificar. Para estructuras de datos más complejas, buscar un vértice cualquiera en toda la cola podría resultar muy costoso. Resulta *ideal* tener acceso directo a cualquier vértice de la cola.

En el capítulo III se describen los *TDA's* para el algoritmo de DIJKSTRA, en particular se tiene una estructura, llamada *Ruta*, de n *Estados* en la cual se almacena la situación

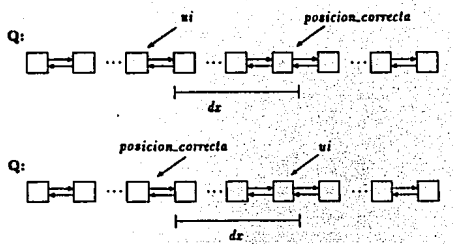


Figura 43: d_x.

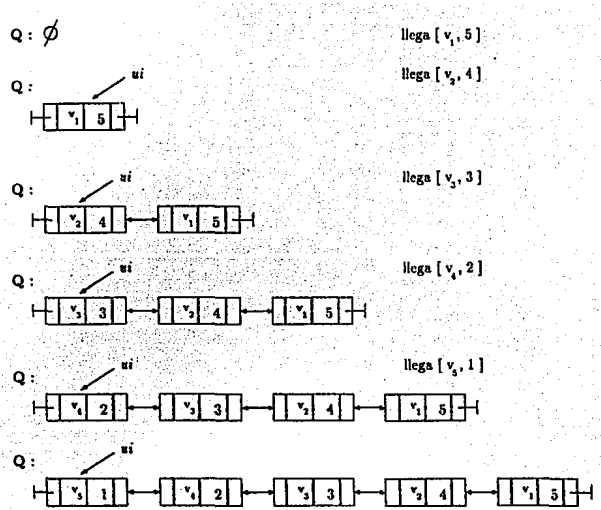


Figura 44: Ejemplo Operación **Inserta**.

actual de cada vértice. Más especificamente, esta estructura es un arreglo de n *Estados*. Cada vértice, con etiqueta temporal finita, puede tener en su *Estado* un apuntador activo a su etiqueta en **Q**. De esta forma, dado un vértice x, la manera de accesarlo en **Q** resulta inmediata. Una vez hecho este cambio, la operación **Decrementa_Llave** se realiza, en general, más rápido:

> Utilizando listas simplemente ligadas la operación se realiza ahora en tiempo costante, las demás operaciones no cambian.
> Llamemos a esta nueva implantación *Clase pq_lsl_ap.*
>
> Si empleamos listas doblemente ligadas que conserven el orden de los elementos según su prioridad en la cola, el tiempo de ejecución de la operación **Decrementa_Llave** depende del proceso que *rubica* al dato en **Q** y tal queda en función del el número de elementos que hay entre x y su posición correcta en **Q** después de haber hecho el decremento, sea d_x tal distancia. Por tanto, **Decrementa_Llave** requiere tiempo $O(d_x)$. Nótese que en el mejor de los casos $d_x = 0$ y en el peor de los casos $d_x = q$, donde q es el tamaño de la cola. Las otras funciones no cambian.
> Denominemos como *Clase pq_ldl_ap* a esta implantación.

Estas clases no alteran el comportamiento observado en los análisis hechos para el algoritmo DIJKSTRASORT, ya que este algoritmo se aplica a gráficas estrellas, las cuales no tienen ciclos y por ello no se efectuan operaciones **Decrementa_Llave**. Pero si mejoran el comportamiento general del algoritmo de DIJKSTRA.

La Tabla 4 muestra el desempeño computacional de las implantaciones concretas utilizando listas ligadas.

Operación / Clase	Tiempo de Ejecución			
	pq_lsl	pq_ldl	pq_lsl_ap	pq_ldl_ap
Inserta	$\Theta(1)$	$O(d_x)$	$\Theta(1)$	$O(d_x)$
EncuentraMin	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
BorraMin	$O(n)$	$\Theta(1)$	$O(n)$	$\Theta(1)$
DecrementaLlave	$O(n)$	$O(n)$	$\Theta(1)$	$O(d_x)$

Tabla 4: Desempeño Computacional para Listas Ligadas

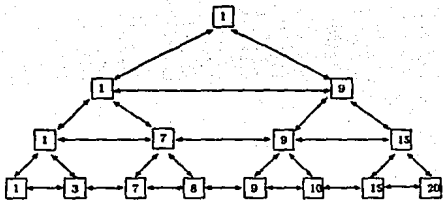


Figura 45: Ejemplo de Level Linked Tree.

2 Level Linked Trees

Los Level_Linked_Trees (LLT) son una generalización de los (a,b)-Arboles, los cuales son usados para representar listas ordenadas [23].

Definición

Se dice que **T** es un **(a,b)-Árbol** si

a) Todas las hojas de **T** tiene la misma profundidad.

b) Para todo nodo v en **T** se tiene que $\delta(v) \leq b$.

c) Para todo nodo v en **T**, excepto la raíz, se tiene que $\delta(v) \geq a$.

d) La raíz r, al menos que sea una hoja, safisface que $\delta(r) \geq 2$.

* El valor de $\delta(v)$ representa el número de hijos que tiene el nodo v.

Mehlhorn [23] nos proporciona el siguiente resultado:

Teorema.- Si un conjunto S de tamaño n es representado por un (a,b)-Árbol, entonces las operaciones **Inserta, EncuentraMin, Elimina y BorraMin** toman tiempo $O(\log_2 n)$.

En un LLT todos los arcos del árbol son transitables en ambas direcciones, entre padres e hijos, además cada nodo tiene apuntadores a sus nodos vecinos del mismo nivel. Si la lista se representa como las hojas de un (a,b)-Árbol, entonces un indicador (finger) es un apuntador a una hoja. Los LLT permiten búsquedas muy rápidas entre las vecindades de los indicadores. La Figura 45 representa un LLT para la lista $\ell = < 1,3,7,8,9,10,15,20 >$.

Se define un **Finger-Tree** como un LLT con apuntadores (indicadores) a algunos de sus hojas.

Descripción de las operaciones

Indicador Es un apuntador a alguna de las hojas del árbol.

Split(y **: dato;** S_1, S_2, S_3 **: Secuencias_Datos)** Separa la secuencia S_1 en dos, a partir de y, esto es: $S_2 = \{x \in S_1 / x \leq y\}$ y $S_3 = \{x \in S_1 / x > y\}$. Finalmente destruye la secuencia S_1. Toma tiempo $O(\log_2(\max\{|S_1|, |S_2|\}))$.

Concatena(S_1, S_2, S_3 **: Secuencias_Datos).** Une dos secuencias de datos, S_1 y S_2 y las deja en S_3. Esta función se lleva a cabo sólo si $\max S_1 < \min S_2$. Finalmente destruye las secuencias S_1 y S_2. Toma tiempo $O(\log_2 |S_1|)$.

Inserta(x **: dato;** S **: Secuencias_Datos)** Utilizando **Concatena**, toma tiempo $O(\log_2(\max\{1, |S|\}))$.

BorraMin(S **: Secuencias_Datos).** Se sigue el apuntador de la raíz al mínimo elemento y se elimina. Caminamos de regreso hacia la raíz re-balanceando el árbol y actualizando el mínimo y los apuntadores sobre el camino. Requiere tiempo $O(\log_2 |S|)$.

EncuentraMinimo. El mínimo se encuentra en la raíz del árbol, así que accesar al mínimo se realiza en tiempo constante.

3 d-Heaps

Se considera un **d-Heap** como un árbol enraizado, para el cual los arcos representan una relación *predecesor-sucesor* para cada nodo[6]. Cada nodo en un d-Heap tiene a lo más d-hijos. La raíz de un árbol es almacenada, utilizando los "indices-predecesores" y los conjuntos de sucesores, como sigue:

$pred(x)$ es el predecesor (padre) del nodo x en el *d-Heap*. El nodo raíz no tiene predecesor.

$SUCC(x)$ es el conjunto de sucesores (hijos) del nodo x en el d-Heap. Un nodo hoja no tiene sucesores.

[6]En esta sección haremos referencia a *nodo* como un elemento del Heap, no como un vértice de la red.

Descripción de las operaciones.[7]

Invariante.- La llave de un nodo i en el **heap** es menor o igual a la llave de cada uno de sus sucesores.

Proceso siftup(i).- Intercambia i con su predecesor mientras i no sea un nodo raíz y la llave de i sea menor que la llave de su predecesor. El proceso queda descrito de la siguiente forma:

Mientras i no sea un nodo raíz y $\mathbf{LLave}(i) < \mathbf{LLave}(Pred(i))$
hacer $Intercambio(i, Pred(i))$

En otras palabras, **siftup** reubica un nodo en el *d-Heap*, buscando su posición de *abajo* hacia *arriba*. Requiere tiempo $O(\log_d n)$

Inserta.- El nuevo elemento i se agrega en la última posición del heap. Después se aplica el proceso **siftup(i)**. Requiere tiempo $O(\log_d n)$.

Proceso siftdown(i).- Sea $MinCh(k)$ el nodo con la menor llave en SUCC(k). Intercambia i con $MinCh(i)$, el nodo con la menor llave en SUCC(i), mientras i no sea un nodo hoja y la llave de i sea mayor que la llave de MinCh(i).

Mientras i no sea un nodo hoja y $\mathbf{LLave}(i) > \mathbf{LLave}(MinCh(i))$
hacer $Intercambio(i, MinCh(i))$

En otras palabras, **siftdown** reubica un nodo en el d-Heap, buscando su posición de *arriba* hacia *abajo*. Requiere tiempo $O(d \cdot \log_d n)$.

EncuentraMin.- El nodo raíz del Heap contiene al elemento mínimo. Requiere tiempo constante, $\Theta(1)$.

BorraMin.- Por construcción, el nodo i es el nodo raíz del Heap. Sea j el nodo almacenado en la última posición del arreglo. Se ejecuta un $Intercambio(i, j)$, se elimina i, después se aplica un $siftdown(j)$ para restaurar el orden en el Heap. Se realiza en tiempo $O(d \cdot \log_d n)$.

Decrementa_Llave.- Decrementa la llave de i y después aplica el proceso $siftup(i)$ para restaurar el orden del Heap. Requiere tiempo $O(\log_d n)$.

Llamemos a esta implantación concreta *Clase pq_dh*. La Tabla 5 nos muestra los tiempos de ejecución de las operaciones para esta implantación, para un **d-heap** en general y para un heap binario (d=2).

[7]Implantación descrita por Ahuja, Manganti y Orlin [1], en la cual simulan el heap en un arreglo

Operación / Clase	Tiempo de Ejecución	
	pq_dh	HeapBinarios
Siftup	$O(\log_d n)$	$O(\log_2 n)$
Siftdown	$O(d \cdot \log_d n)$	$O(\log_2 n)$
Inserta	$O(\log_d n)$	$O(\log_2 n)$
EncuentraMin	$O(1)$	$\Theta(1)$
BorraMin	$O(d \cdot \log_d n)$	$O(\log_2 n)$
DecrementaLlave	$O(\log_d n)$	$O(\log_2 n)$

Tabla 5: Desempeño computacional de los d-Heaps.

4 Colas Binomiales

Esta estructura de datos soporta todas las operaciones elementales de las colas de prioridades. Es presentada en 1978 por J. Vuillemin [47].

4.1 Especificaciones

Las **Colas Binomiales** están basadas en un árbol de singular estructura, denominado **árbol binomial** B_r, el cual se define por inducción de la siguiente manera:

- B_0 es un único nodo y
- B_r es creado por la unión de dos copias de B_{r-1}

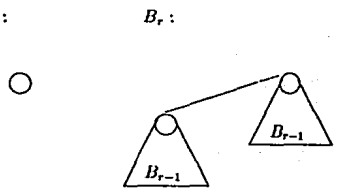


Figura 46: Definición de Árbol Binomial.

La Figura 46, ilustra la construcción de un árbol binomial. Un árbol binomial es un árbol de la clase B_k, para algún entero k. A k se le denomina **indice** del árbol binomial.

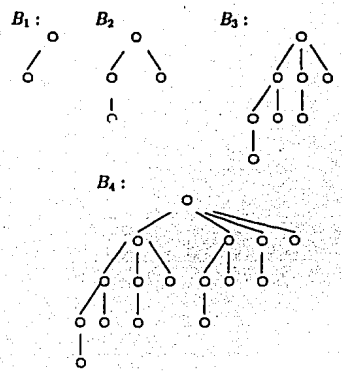


Figura 47: Primeros arboles binomiales.

Para ejemplificar un poco más, en la Figura 47 se ilustran los cuatro primeros arboles binomiales. Una forma alternativa de construir un árbol binomial B_k se muestra en la Figura 48.

Mark Brown [34] prueba las siguientes propiedades:

- ⋆ El árbol B_r tiene 2^r nodos.
- ⋆ La altura del árbol está dada por: $h(B_r) = r + 1$.
- ⋆ La raíz tiene exactamente r hijos.
- ⋆ B_r contiene en el nivel i exactamente $\binom{r}{i}$ nodos, combinaciones de r en i nodos, el cual es el *Coeficiente Binominal.*

Definición

Una Cola Binomial es un bosque de arboles binomiales heap-ordenados.
El total de nodos en la cola binomial, representa su tamaño.

Para facilitar la presentación de las colas binomiales, en este anexo, los apuntadores a las raices de los arboles estrán representados en un arreglo, aunque pueden ser presentados en una lista ligada. El tamaño de los arboles binomiales está determinado por la representación binaria de n, el número de objetos requeridos en el bosque.

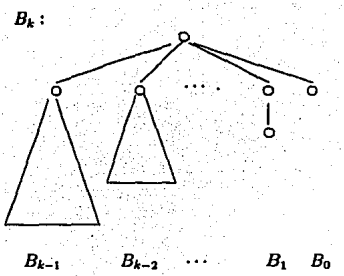


Figura 48: Construcción Alternativa, Arboles Binomiales.

Ejemplo

Suponga que $n = 9 = (1001)_2$, esto nos indica que se requieren dos arboles binomiales: B_0 y B_3. La representación gráfica está dada por la Figura 49.

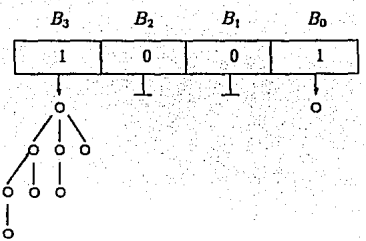


Figura 49: Ejemplo: Cola Binomial ($n = 9 = (1001)_2$).

Descripción de las Operaciones.

MinH.- Apuntador al Nodo Mínimo.

Une (T_1, T_2 : Árbol_Binomial).- [Merge] Dados dos arboles binomiales heap-ordenados con el mismo indice k, genera un nuevo árbol binomial heap-ordenado

con indice $k+1$ para el cual la raíz será el nodo raíz que tenga menor llave. Es decir, las llaves de la dos raices se comparan y la más chica tendrá como descendiente directo a la mayor. Esta operación requiere tiempo constante.

Funde (CB: Cola_Binomial).- [Funde] Une la cola binomial CB con $\mathbf{Q}$ y deja la cola resultante en $\mathbf{Q}$. Esta operación supone que existan arboles binomiales con igual indice en $\mathbf{Q}$ aplica la operación **Une**. Requiere tiempo $O(\log n)$, donde n es el número de elementos en la cola resultante.

Inserta (i).- Crea una cola binomial X consistente de un sólo objeto, el nodo i. Después efectúa **funde**(X). Requiere tiempo $O(\log n)$.

EncuentraMin.- Regresa el 'valor' de $MinH$. Requiere tiempo $\Theta(1)$.

BorraMin.- Primero se asigna $i \leftarrow MinH$. Después elimina el nodo mínimo, esto incrementa el número de arboles en la cola, los cuales hay que reorganizar aplicando **funde**. Requiere tiempo $O(\log n)$.

DecrementaLLave(i,k) .- Decrementa la llave de x y después aplica el proceso $siftup(x)$ para restaurar el orden del árbol. Toma tiempo $O(\log n)$.

Siftup(x).- Intercambia x con su predecesor mientras x no sea un nodo raíz y la llave de x que la llave de su predecesor. Requiere tiempo $O(\log n)$.

Denominemos a esta implantación concreta, *Clase pq_bq*. La Tabla 6, resume el desempeño computacional para las operaciones de la clase *pq_bq*.

Clase **pq_bq**	
Operaciones	Tiempo
Une, EncuentraMin	$\Theta(1)$
Funde, Inserta, BorraMin, DecrementaLlave	$O(\log_2 n)$

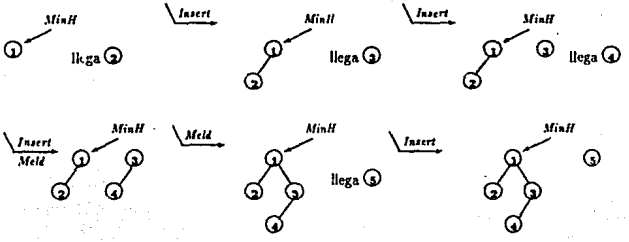
Tabla 6: Desempeño Computacional de la Clase *pq_bq*.

A continuación se desarrolla un ejemplo, que nos ilustre el comportamiento de las operaciones anteriores para ver si estas se pueden mejorar.

Ejemplo

Suponga que $\ell = \{1, 2, 3, 4, 5\}$, la Figura 50, muestra, paso a paso, la forma como se ejecuta el Algoritmo DIJKSTRAM para ℓ.

Inserciones:



Eliminación de mínimos

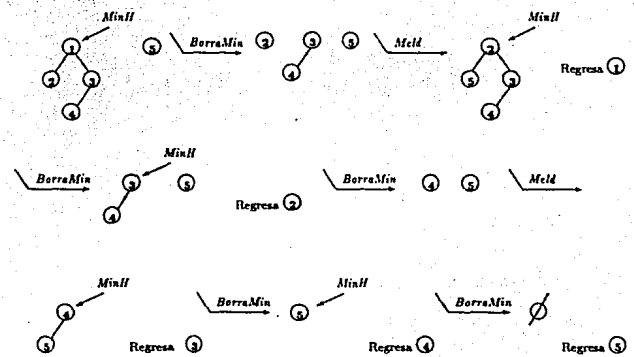


Figura 50: Ejecución del Algoritmo DIJKSTRAM

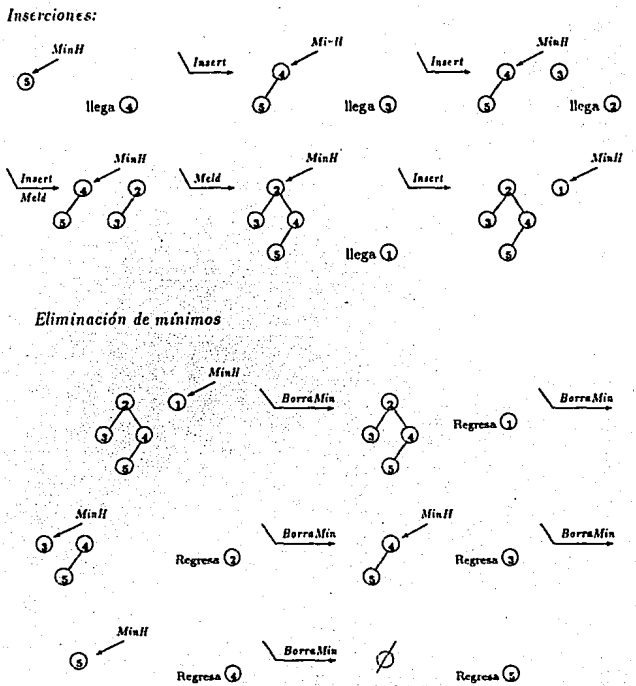


Figura 51: Otra ejecución del DIJKSTRAM

Al parecer, el algoritmo no aprovecha que la secuencia ℓ ya se encuentra ordenada. Supongamos que al guardar ℓ en la *Lista de Espacecia* en vez de almacenarlos como en una *cola* los iremos *apilando*. El algoritmo se efectúa como lo muestra la Figura 51.

Nótese que. **BorraMin** no aplica **Meld's** y por la forma como van llegando los datos, el mínimo siempre queda en el árbol más pequeño.

4.2 Estructuras de Datos

Estructura V.

Vuillemin suguiere en 1978 [47] una estructura de Datos para manipular las colas binomiales, la cual es conocida como **Estructura V.**

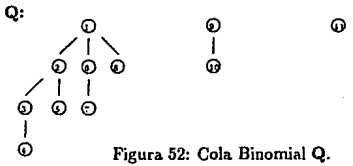


Figura 52: Cola Binomial Q.

La explicación de ésta se ilustrará con un ejemplo. Considerese la Cola Binomial Q mostrada en la Figura 52. La representación V de Q es ilustrada en la Figura 53. Es decir, se tiene una lista para las raices. Cada raíz apunta a uno de sus hijos. Los hijos a su vez están en una lista ligada. Cada hijo apunta a uno de sus hijos.

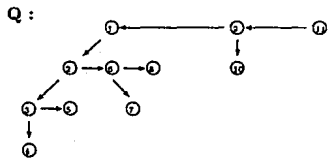


Figura 53: Representación V de Q.

Esta estructura tiene un pequeño problema: no es fácil apreciar que un hijo *no* forma parte de la lista de padres. El ejemplo de la Figura 53 está tramposamente estructurado, bien pudo dibujarse como en la Figura 54 y con tal representación no sabríamos si el *nodo* 2 forma parte de la lista de padres, esto es, que sea hermano del *nodo* 1, o si es hijo del *nodo* 1.

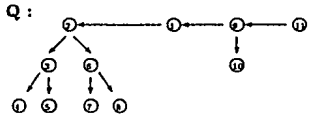


Figura 54: Otra representación V de **Q**.

Estructura Ring (Estructura R).

Esta estructura organiza a los nodos de un mismo nivel (los que son hermanos o las raices) en una lista circular. La representación de **Q** usando la **Estructura R** se muestra en la Figura 55. En esta representación ya no se tiene el problema de confundir a los hermanos o padres con los hijos.

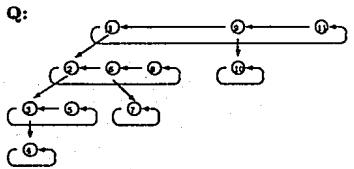


Figura 55: Estructura R de **Q**.

Para cada una de las estructuras se recomienda tener un identificador de la cola, denominado **Encabezado o Cabeza**, el cual puede tener dos campos: **Tamaño**, representación binaria del número de nodos y **Cola**, apuntador a la Cola Binaria. A continuación se representan los diagramas de diferentes estrucuras que, según Brown [34], permiten cualquier tipo de eliminación, Figura 56 a Figura 61.

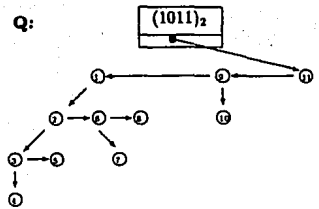


Figura 56: Estructura V.

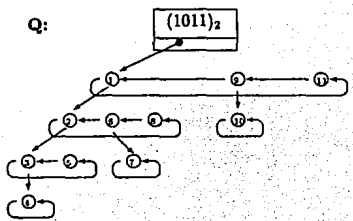


Figura 57: Estructura R.

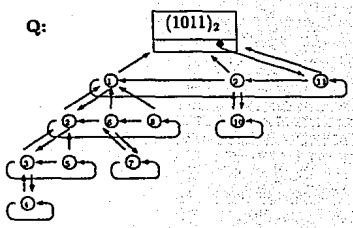


Figura 58: Estructura V con apuntadores hacia arriba.

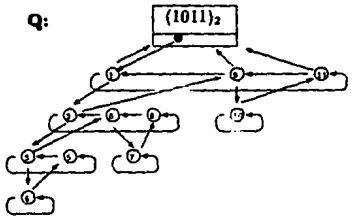


Figura 59: Estructura R con apuntadores hacia arriba.

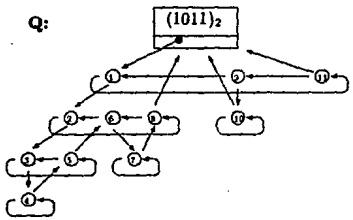


Figura 60: Estructura con sólo dos apuntadores por nodo.

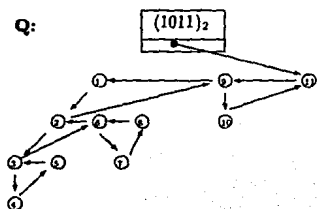


Figura 61: Estructura K.

5 Fibonacci Heaps

Michel L. Fredman y Robert E. Tarjan, en 1987 [31], presentan una *nueva estructura de datos* denominada **Fibonacci Heaps**, la cual mejora el desempeño computacional de los Algoritmos de Optimización en Flujo en Redes, en particular el problema RMC se modifica enormemente.

5.1 Especificaciones

Un **F-Heap** es un bosque de arboles Heap-Ordenados, los cuales *no* requieren de una forma o tamaño especial, es decir, no hay condiciones sobre el número de arboles o su estructura, las únicas restricciones que existen son acerca de la manera como se manipulan los arboles.

En un **F-Heap** sucede que:

- Cada nodo puede o no puede estar **marcado**. Un nodo raíz nunca será marcado.
- Las raíces de los arboles están en una *Lista Circular Doblemente Ligada*.
- Cada nodo tiene un apuntador a uno de sus hijos.
- Los hijos de cada nodo están en una *Lista Circular Doblemente Ligada*. Es decir, cada nodo está en una *Lista Circular Doblemente Ligada* con sus *hermanos*.
- Cada nodo contiene un apuntador a su padre o a *NIL* (los nodos raíz, no tienen padre).
- Cada nodo contiene su *rank* y un bit que indica si está o no *marcado*.
- Existe un apuntador a la raíz que tiene llave mínima, a tal se le llama **Nodo Mínimo**. Se accesa al **F-Heap** por el **Nodo Mínimo**.
- Si el **Nodo Mínimo** es NIL, entonces el **F-Heap** es vacío.

La Representación de los F-Heaps requiere espacio para cada nodo de:

- Cuatro apuntadores: *Padre, Hijo, Hermano Antecesor, Hermano Sucesor*
- Un Entero : *Rank*
- Un Bit : *Marcado??*

Ejemplo: Considerese el bosque, mostrado en la Figura 62, la representación de **B** como un F-Heap se muestra en la Figura 63.

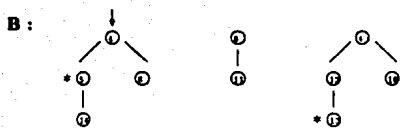


Figura 62: Ejemplo de bosque.

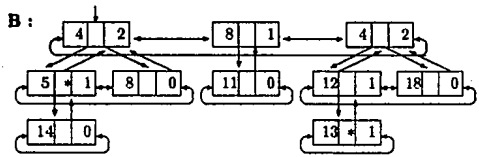


Figura 63: Ejemplo de F-Heap.

5.2 Descripción de las Operaciones

Inserta.- Agraga un nuevo elemento al Heap. Primero crea un nuevo heap consistente del nodo i. después une los dos Heaps con el proceso Funde. Por construcción, del proceso Funde, esta operación gasta tiempo $\Theta(1)$.

Funde.- Une dos Heaps en uno nuevo. Combina las raices de los heaps en una lista sencilla y regresa como *Nodo Mínimo* del nuevo heap a la llave menor de las dos raices, o a cualquiera de ellas en el caso de que sean iguales. Esta Operación gasta tiempo costante.

EncuentraMin.- Regresa el nodo cuya llave es la mínima del Heap. Por construcción, el **F-Heap** tiene un apuntador al *Nodo mínimo*, por consiguiente esta función gasta tiempo $\Theta(1)$.

Link.- Dados dos arboles con el mismo rank, r, crea un nuevo árbol con rank $(r+1)$, donde el nodo-padre será el nodo-raíz con menor llave.

BorraMin.- Almacena en una variable auxiliar el *Nodo Mínimo*, lo elimina de la lista de raices, integra los hijos del *Nodo Mínimo* a la lista de raices, finalmente, *reorganiza* los arboles del F-Heap, utilizando la operación **Link.** Al salir de este proceso, no debe quedar más de un árbol con el mismo rank.

Para reorganizar el F-Heap utilizamos un Arreglo donde los indices representan los diferentes *ranks*. Cada posición del arreglo es un apuntador a la raíz de un árbol. Inserta cada árbol del F-Heap en el Arreglo y los va ligando como van llegando, esto significa que en cualquier momento que se intente insertar una raíz en una posición ya ocupada, se ejecuta, entonces, la operación **link** y se reintenta insertar la raíz del nuevo árbol en la siguiente posición. Después de que se han insertado todas las raices, se *re-construye* el F-Heap, partiendo de que es *NIL*, se recorre el arreglo insertando cada raíz en el F-Heap. De esta forma el arreglo queda vacio y el F-Heap queda bien ensamblado. Finalmente, se regresa el que fuera el nodo mínimo. R. Tarjan y M. Fredman [31], muestran que esta operación gasta en total tiempo amortizado de $O(\log_2 n)$.

DecrementaLlave.- Decrementa la llave de i en Δ unidades. Si la llave a decrementar es una raíz, r, entonces no suceden cambios extras en el Heap, al menos que la nueva llave de i sea más pequeña que la llave del *Nodo Mínimo*, en este caso, se define i como el nuevo *Nodo Mínimo*. Si la llave a modificar no es una raíz, efectúa el proceso **CascadeCut.**

CascadeCut.- Dado un nodo i, quita la liga que une a i con su padre, $p(i)$, garantiza que i quede desmarcado, decrementa en uno el rank de $p(i)$, padre de i, elimina a i de la lista que lo une con sus hermanos, añade a i en la lista de raices del F-Heap. Finalmente,

Si $p(i)$ **está marcado** **DesMarcar** al $p(i)$ y aplicar **CascadeCut** a $p(i)$.

En otro caso Si $p(p(i))$ **no es una raiz** Marcar al $p(i)$.

Elimina.- Quita del F-Heap un elemento arbitrario, i. Dado un vértice i para eliminarlo del F-Heap. Localiza al nodo x que contiene a i, corta la liga que une a x con su padre, forma una nueva lista de raices al concatenar la lista de hijos de x en la lista original de raices, finalmente, destruye x.

5.3 Observaciones

+ Una simple llamada a un **DecrementaLlave** puede provocar un gran número de cortes en cascada (**CascadeCut**), la Figura 64 ilustra este hecho.

+ El propósito de **marcar** los nodos es guardar la huella por dónde se harán los Cortes en Cascada.

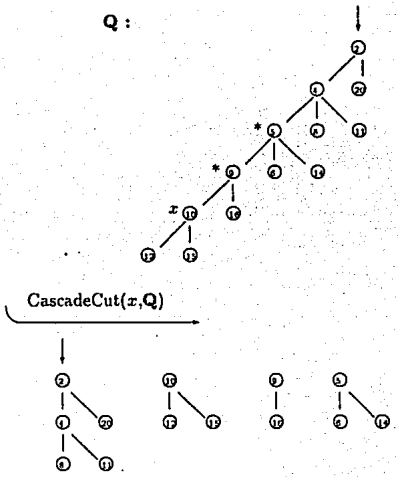


Figura 64: Cortes en Cascada.

+ El análisis dado por Tarjan y Fredman [31] sobre los F-Heaps depende de dos propiedades cruciales:

  1. Cada árbol en un F-Heap no necesariamente es un árbol binomial, pero tiene un tamaño al menos exponencial en el *rank* de su raíz.
  2. El número de *Cortes en Cascada* que acontece durante una secuencia de operaciones Heap es acotado por el número de operaciones que realizan **DecrementaLLave** y **Elimina**.

+ Los Cortes en Cascada se introdujeron en la manipulación de los heaps para preservar la propiedad 1. Además la condición para que ello ocurra, es la llamada Regla de la Pérdida de dos hijos la cual limita la frecuencia de los **CascadeCut** descrita en la propiedad 2.

+ La propiedad 1 está basada en la siguientes afirmaciones:

  * **Lema.-** Sea x cualquier nodo en un F-Heap. Acomodar los hijos de x en el orden

en que estos fueron ligados a x. Entonces, el i-ésimo hijo de x tiene un rank de al menos $(i-2)$.

* **Corolario.-** Un nodo con rank k en un F-Heap tiene al menos: $F_{k+2} \geq \phi^k$ descendientes, incluyendo el mismo. Aqui F_k es el k-ésimo número de Fibonacci: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$, $k > 3$ y $\phi = (1 + \sqrt{5})/2$.

* **Teorema.-** Sea x un nodo en un F-Heap. Sea $k = rank(x)$. El tamaño del árbol enraizado en x satisface: $k \geq F_{k+2}$.

Los arboles con el mínimo tamaño posible dado un rank en un F-heap son mostrados en la Figura 65, la cual además nos ilustra el Teorema anterior.

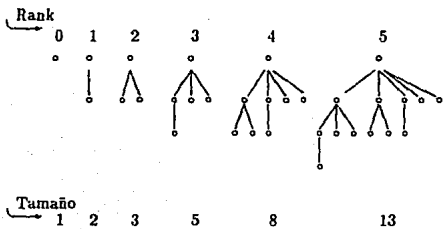


Figura 65: Árboles con el mínimo tamaño posible.

C Programación Orientada a Objetos

1 Introducción

La **Programacion Orientada a Objetos (POO)** ha adquirido gran importancia en los últimos años y las definiciones de los conceptos más importantes dependen, en cierto sentido, del punto de vista de cada autor y del lenguaje de programación orientado a objetos (LOO).

Este apendice pretende dar una definición de los conceptos más comunes de la POO sin particularizar en algún lenguaje de programación. Después describiremos, brevemente, algunas de las especificaciones del LOO **Eiffel**.

2 Conceptos Básicos de la POO

Abstracción.- "... es un proceso mediante el cual las entidades se caracterizan por propiedades de interés para un fin específico" [3]. La abstracción hace énfasis en los principales detalles de lo que se desea modelar. Identifica los objetos que interesan y destaca sus caracteristicas esenciales. La abstracción depende del observador.

Ejemplo: A un automóvil el conductor lo *ve* como un medio de transporte cómodo; un mécanico lo *ve* como algo que esta integrado por motor, radiador, carburador, etcétera; un ingeniero diseñador de autos, lo *ve* como un algo a optimizar: mejorar el sistema de locomoción, sistema de frenado, sistema hidráulico, sistema de calefacción, carrocería, etcétera.

La abstracción tiene propiedades estáticas y dinámicas. Las primeras no permiten que el objeto cambie en cuanto a su funcionamiento. Las propiedades dinámicas son el cambio en el valor de las propiedades estáticas.

Ejemplo: En un automóvil, se tiene propiedades estáticas como motor, carrocería, llantas, rines, radiador, tanque de gasolina. Una propiedad dinámica será cambiar cualquiera de las propiedades anteriores, como cambiar una llanta o los rines.

Encapsulación.- "... es el proceso de esconder todos los detalles de un objeto que no contribuyen a sus caracteristicas esenciales" [3]. La encapsulación es un concepto complementario de la abstracción; la primera se enfoca en la vista externa de los objetos y la encapsulación impide que se vea su parte interna.

Ejemplo: Una radio tiene como característica esencial transmitir sonidos, sin embargo la mayoría de las personas desconocemos como la radio capta las ondas hertzianas; así

pues, podemos decir que este proceso se encuentra encapsulado.

Modularidad.- "... es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos con un alto grado de cohesión y ligeramente acoplados entre sí" [3]. Entendemos por módulo a un conjunto de elementos lógicamente relacionados.
Ejemplo: A un automóvil lo podemos dividir en el Sistema Hidráulico, Sistema de Frenado, Calefacción, Sistema de Locomoción, entre otros y cada uno de ellos podemos verlo como un módulo.

Jerarquía.- "... es una clasificación de las abstracciones" [3]. Mediante las jerarquías se organizan los objetos en distintos niveles de abstracción.

Concurrencia.- "... es la propiedad que distingue a un objeto activo de uno no activo" [3]. La concurrencia permite que diferentes objetos actuen al mismo tiempo.
Ejemplo: En un automóvil, el sistema de locomoción y la calefacción pueden estar activos al mismo tiempo, cuando eso sucede se dice que son concurrentes. Otro ejemplo, en un automóvil el radiador y la bateria son objetos concurrentes.

Persistencia.- "... es la propiedad de un objeto a través de la cual su existencia trasciende en tiempo y/o espacio" [3]. Se dice que un objeto es **persistente** si no es destruido cuando finaliza la ejecución del sistema.

3 Conceptos Generales de la POO

Objetos.- En términos generales, concebimos un objeto como una cosa, un elemento, una forma. Desde el punto de vista de los lenguajes de programación y a nivel elemental un objeto es un conjunto de propiedades y métodos para operarlas. Un objeto puede contener otros objetos, pero un objeto interno no puede ser compartido por otros objetos. Compartir objetos es necesario cuando varios objetos deben hacer uso de información común, esto no sólo economiza memoria sino que los cambios que se hacen en el objeto que se está compartiendo se reflejan simultaneamente en todos los objetos que están haciendo uso de él, evitando problemas de inconsistencia.
Ejemplo: Un automóvil está constituido por otros objetos como: sistema de arranque, sistema de frenado, el radiador, la suspensión, entre otros.

En la POO, los objetos tienen estado, comportamiento e identidad. El estado de un objeto lo determina un conjunto de características determinadas por las propiedades

estáticas y el valor de las mismas. El **comportamiento**, se refiere al funcionamiento del objeto: *para qué sirve*. La **identidad** son las características particulares que lo distinguen de otros objetos similares.

Ejemplo: En un automóvil el estado está determinado por el motor, el radiador, la transmisión, el sistema de frenado, las llantas, entre otros; al ponerlo en marcha uno puede transladarse de un lugar a otro, ese su funcionamiento; la identidad del automóvil está determinada por su número de placas, el cual es único para cada auto.

Los objetos por si solos no existen en un sistema, requieren de las relaciones de uso que hay entre ellos. Los objetos, sirven como operandos sobre los cuales pueden efectuarse operaciones y como operadores que realizan operaciones sobre otros. Al conjunto de operaciones que un objeto puede realizar sobre otro se le llama **protocolo**.

Dependiendo de la función que realice un objeto sobre otro, un objeto puede ser actor, servidor o agente. Un **actor** es un objeto que opera sobre otros, pero ningún objeto puede operar sobre él. Un **servidor** es un objeto que nunca opera sobre otros, pero otros objetos si pueden operar sobre él. Un **agente** es un objeto que puede operar y ser operado por otros objetos.

En la programación estructurada cuando un procedimiento hace uso de otro procedimiento o función se dice que está haciendo una invocación a un proceso, en la POO cuando un objeto llama a otro se denomina: **envío de mensaje.**

Clases.- Una **clase** es una estructura que agrupa un conjunto de objetos que tienen las mismas características en cuanto a estructura y funcionamiento. "Las clases pueden verse como moldes para formar objetos[11]."

Los objetos son elementos que deben ser creados durante la ejecución de un sistema, las clases son descripciones estátaticas de un conjunto de posibles objetos, ejemplares de la clase. En tiempo de ejecución se tienen únicamente objetos, en el programa sólo se observan clases.

No todos los componentes de una clase están disponibles para cualquier otra clase y para los usuarios se dividen en dos partes, una interna y otra externa. En la parte externa, llamada **interfaz de la clase**, se declaran todas las operaciones que se aplican a los ejemplares de la clase. En la parte interna se implantan todas las funciones que fueron declaradas en la interfaz. Las operaciones que se aplican a los ejemplares de una clase se denominan **métodos**, el conjunto de métodos es el protocolo.

La interfaz de una clase está divida en tres partes: la pública, la protegida y la privada. **La parte pública** es visible y accesible para cualquier usuario de la clase. **La parte protegida** son declaraciones que no pueden ser usadas por cualquier clase; sólamente las subclases de una clase tienen acceso a la parte protegida. **La parte privada** es la representación del objeto, las declaraciones que se encuentran en esta parte sólo pueden ser usadas por los métodos de la clase.

La diferencia esencial, entre una subclase y un usuario de la clase es que el usuario de la clase crea ejemplares de la misma y envía mensajes a esos objetos, mientras que una subclase crea nuevas clases basándose en la clase de la cual proviene.

Creación, Iniciación y Destrucción de objetos.- La **creación** implica destinar espacio en memoria para contener un nuevo objeto y asociar ese espacio con un nombre. Hay dos tipos de variables: dinámicas y automáticas. Una **variable dinámica** es aquella en la que el espacio que va a ocupar en memoria debe ser apartado explicitamente por el programador. Una **variable automática** es aquella que se crea cuando se encuentra su declaración en un procedimiento y el espacio que ocupa es liberado cuando éste termina. La **iniciación** es el proceso que da valores iniciales a los objetos. La **destrucción** de un objeto puede ser automática, cuando el objeto ya no se usa; o puede llevarse a cabo mediante funciones que proporcione el lenguaje o que cree el programador.

Herencia Sencilla.- Las clases se relacionan de manera jerárquica. Al agrupar en una clase un conjunto de objetos con características y funcionamientos similares, de ella puede irse derivando nuevas clases (subclases) que **hereden** sus propiedades. Estas nuevas clases son más específicas. Una **superclase**, en general, es toda clase que hereda su funcionamiento a otra clase.
Ejemplo: La clase *Automovil*, puede derivar las subclases *Auto Compacto*, *Auto Deportivo*, *Auto de Carreras*. A su vez, la clase *Automovil*, pertenece a la superclase denominada *Medios de Transporte*, para la cual además se tienen clases como *Ferrocarriles*, *Aeronaves*, *Barcos*, etcétera. La figura 66 ilustra la jerarquía de la Clase *Medios de Transporte*.

Ventajas de la Herencia

a) **Reutilización de *Software*.** Los métodos de una superclase son heredados, por lo cual, en general, no hay que re-escribir o re-definir.

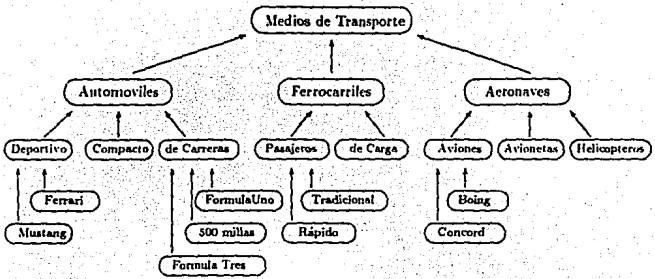


Figura 66: Herencia Sencilla

b) **Código Compartido.** Sucede cuando varios programadores usan la misma clase.

c) **Consistencia en la Interfaz.** Debido a que varias subclases heredan funcionamiento y propiedades de una misma superclase, se está seguro de que el funcionamiento de los ejemplares de las clases herederas es similar.

d) **Ocultamiento de Información.** Cuando un usuario usa una clase, sólo requiere saber cómo funciona, por ello únicamente se le proporciona la interfaz y no los detalles de implantación.

e) **Polimorfismo.** Una función puede ser usada con una variedad de argumentos. Esto permite que el código sea escrito una sola vez, en una abstracción de alto nivel, y luego sea ajustada a varias situaciones.

Herencia Múltiple.- Se dice que hay **Herencia Múltiple** si una clase **A** hereda las propiedades de dos o más clases. Usando el mecanismo de herencia se va de un nivel general a un nivel particular.

Ejemplo: Consideremos la Clase *ChoferFórmulaUno*, el cual tiene como superclase a la Clase *Chofer*, que a su vez pertenece a la superclase *Personas*, la cual a su vez pertenece a la clase *Mamiferos*. Por otra parte consideremos la Clase *Medios de Transporte*, la cual es descrita en la Figura 66. Para *definir* la Clase *EscuderíaFórmulaUno* requerimos de un *ChoferFórmulaUno* y de un *Auto de Carreras de FórmulaUno*; esto es, debemos heredar las propiedades de la Clase *ChoferFórmulaUno* y *Auto de Carreras*

de Fórmula Uno. La figura 67 nos ilustra gráficamente este ejemplo.

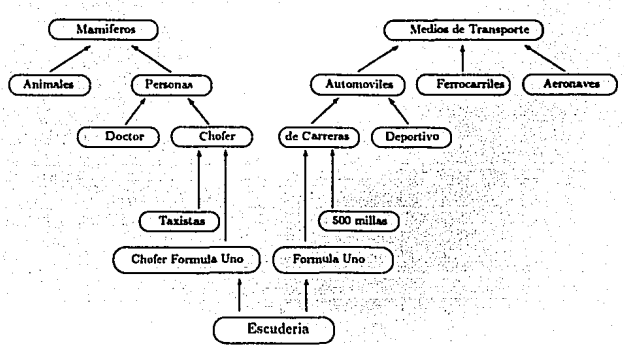


Figura 67: Herencia Múltiple

Asociación Estática y Dinámica.- La **asociación** es el significado que se da a los atributos en alguna parte del programa, es el sentido de una construcción particular. La asociación puede hacerse durante la compilación, donde el código del programa es encontrado por primera vez; durante el ligado, cuando los resultados de las diferentes compilaciones se combinan; y en tiempo de ejecución.

En la **asociación estática** los tipos de todas las variables y expresiones se fijan durante la compilación. La asociación de un mensaje con un método está basada en las características estáticas de las variables. En la **asociación dinámica** los tipos de todas las variables y expresiones se conocen hasta la ejecución del sistema. Esta asociación y el polimorfismo estan muy relacionadas ya que en las funciones polimórficas el significado que se les asocia se sabe hasta que el sistema se ejecuta.

Polimorfismo Polimorfismo significa "muchas formas". En términos de lenguajes de programación, un **objeto polimórfico** es cualquier entidad que permite valores de diferentes tipos durante la ejecución de un sistema. Una **función polimórfica** es aquella que tiene argumentos polimórficos.

En los lenguajes de programación la forma más común de polimorfismo es la **sobre-**

carga (*overloading*): el símbolo + es un símbolo sobrecargado, ya que puede hacer la suma de reales y enteros, en algunos lenguajes se utiliza para concatenar caracteres (cadenas de caracteres).

El **Polimorfismo Puro** ocurre cuando a una función simple se le pueden aplicar argumentos de diferentes tipos. En el polimorfismo puro se tiene una función y un número de interpretaciones diferentes. La **sobrecarga** es otro tipo de polimorfismo, también es conocido como **polimorfismo ad-hoc**, sucede cuando se tiene un número de funciones diferentes denotadas por el mismo nombre, el código a ejecutar depende de los argumentos que se den.

D Implantación del Dijkstra con Radix-Heaps

En este anexo, trabajaremos con un modelo de cómputo, denominado **Arboles Racionales de Decisión** (Rational Decision Trees) [23], el cual es muy diferente al modelo que habiamos estado manejando. En el modelo de Arboles Racionales de Decisión, podemos comparar funciones racionales de los ejemplares. Esto es, podemos hacer operaciones sobre los bits de los datos de entrada.

1 Generalidades del Radix Heaps

Los **R-Heaps** son un tipo de datos concreto del *TDA Cola de Prioridades*. Fueron descritos por Ahuja, Magnanti y Orlin [2]. Los R-Heaps están formados por *Canastas (Buckets)* en las cuales se almacena a los nodos con etiquetas temporales finitas. Considerando que la ruta más larga de s a t utiliza, en el peor de los casos, n nodos y $(n-1)$ aristas, la longitud de esa ruta tendrá costo acotado por nC, donde C es el valor máximo de todos los costos de la red.

Se define $K = \lceil \log_2(nC) \rceil$ y se utilizan únicamente $(K+1)$ canastas, enumeradas como: $0, 1, \ldots, K = \lceil \log_2 nC \rceil$. Se define $\mathbf{P}_{max}$ como la etiqueta del último nodo etiquetado como *permanente*, es decir: $\mathbf{P}_{max} \leftarrow \min \{d(i) | i$ tiene etiqueta permanente$\}$. Para cada canasta se establecen rangos **L** y **H** que dependen de la representación binaria de $\mathbf{P}_{max}$, la cual denotaremos por $(\mathbf{P}_{max})_2$.

El análisis de una implantación R-Heap confia en la evaluación directa del desempeño computacional de los procesos de **BuscaMin** y **Actualiza**[2]. A continuación se describen estos procesos.

Regla de Almacenamiento.- Un nodo v se almacena en la canasta B_j siempre que: $\mathbf{L}[j] \leq d(v) \leq \mathbf{H}[j]$ $(\leq \mathbf{L}[j+1])$. Pero tomando en cuenta que si las canastas se representan en una *Lista Doblemente Ligada*, el acceso a la canasta B_j, no es inmediato, se debe recorrer secuencialmente la lista hasta encontrar tal B_j.

Proceso BuscaMin.- Hace una búsqueda secuencial sobre las canastas, desde B_0 hasta encontrar la primera canasta no vacía, B_B. Después busca secuencialmente en B_B al nodo v cuya etiqueta sea la miníma. $\mathbf{P}_{max}$ será ahora $d(v)$, modifica

los rangos **L** y **H**: finalmente reubica los nodos de B_B, utilizando la **Regla de almacenamiento.**

Rango L.- Se define el rango $L[j]$, L_j, como el mínimo número que pueda formarse al variar los primeros j-bits, de derecha a izquierda, de $(P_{max})_2$, quedando en el j-ésimo bit en un 1 *fijo*. De esta manera, $L[j]$ queda constituido de la siguiente forma:

+ L_j y P_{max} coinciden en los bits $K, (K-1), \ldots, (j+1)$.
+ El j-ésimo bit de L_j tiene un 1.
+ L_j tiene 0's del bit $(j-1)$ al bit 0.

El Rango H.- Se define a $H[j]$, H_j, como el número más grande que pueda formarse al variar los primeros j-bits, de derecha a izquierda, de $(P_{max})_2$. Es decir queda formado de la siguiente manera:

+ $H[j]$ y P_{max} coinciden en los bits $K, (k-1), \ldots, (j+1)$
+ $H[j]$ tiene únicamente 1's del bit j al bit 0.

Ahuja, Magnanti y Orlin [2], realizan las siguientes observaciones:

Observación 1.- Las etiquetas de distancia que el Algoritmo de DIJKSTRA designa como permanentes son *no* decrecientes.

Observación 2.- ▷ Si $P_{max} \leq d(i) \leq d(j)$ entonces, el vértice i está en la misma canasta que el vértice j, o bien, el vértice i está en una canasta anterior (con indice menor).

▷ Si el i-ésimo bit de P_{max} es '1' entonces la canasta i está vacía.

Observación 3.- Suponga que el nodo con mínima etiqueta temporal está en la canasta B, después de que el algoritmo actualiza P_{max} y efectua la **Regla de Almacenamiento**, cada nodo en B será trasladado a una canasta con indice menor. Todos los demás nodos permanecen en las canastas que tenían antes de ejecutar **BuscaMin.**

Observación 4.- Cuando un nodo con etiqueta temporal se mueve de una canasta, siempre lo hace a una canasta con índice menor.

Una vez calculada la representación binaria de P_{max}, se puede entonces, calcular los nuevos rangos **L** y **H** para cada canasta y determinar, por medio de una búsqueda secuencial la canasta donde el nodo j será almacenado.

Descripción Detallada del Cálculo para los Rangos L y H

La idea principal es ir variando, de derecha a izquierda, los bits de $(\mathbf{P}_{max})_2$:

- *Rango* 0.– No varían los bits. En la canasta B_0 se almacenan los vértices cuyas etiquetas son iguales a $(\mathbf{P}_{max})$.
- *Rango* 1.– Cambia el bit 1. En la canasta B_1 se almacenan los vértices cuyas etiquetas difieren en el primer bit de $(\mathbf{P}_{max})$.
- *Rango* 2.– Varían los bits 1 y 2. En la canasta B_2 se almacenan los vertices cuyas etiquetas difieren de $(\mathbf{P}_{max})$ en los dos primeros bits.
- *Rango* 3.– Varían los bits 1, 2 y 3. En la canasta B_3 se almacenan los vértices cuyas etiquetas difieren de $(\mathbf{P}_{max})$ en los tres primeros bits.
- ... ...
- *Rango* r.– Varían los bits $1, 2, \ldots r$. En la canasta B_r se almacenan los vértices cuyas etiquetas difieren de $(\mathbf{P}_{max})$ en los r primeros bits.

Ejemplo: Establecimiento de Rangos para las etiquetas

Suponga que: $nC = 63 = (\ 111111\ )_2$ y $P_{max} = 18 = (\ 010010\ )_2$. La Tabla 7 ilustra los rangos.

Rango	L (Lower)	H (High)
0	010010	010010
1	010011	010011
2	∅	∅
3	010100	010111
4	011000	011111
5	∅	∅
6	100000	111111

Tabla 7: Establecimiento de Rangos para $nC = 63 = (\ 111111\ )_2$

2 Algoritmo de Dijkstra con Radix Heaps

En seguida describimos detalladamente el Algoritmo de DIJKSTRA utilizando R-Heaps.

I. Iniciar

1. Sea $K = \lceil \log_2(nC) \rceil$.
2. Asignar las Etiquetas:
   - $d(1) = d(s) = 0$;
   - $d(j) = nC \qquad j = 2, 3, \ldots, n.$
3. Distribuir los n nodos en las $(K + 1)$ canastas.
   - s va a la canasta B_0.
   - $N \setminus \{s\}$ van a la canasta B_K.

II. Mientras haya *Etiquetas Temporales Finitas* en las canastas.

1. Buscar al nodo i con etiqueta mínima.
   a. Buscar una canasta B_η no vacía.
      Recorrer secuencialmente de la canasta B_0 en adelante hasta encontrar una canasta ocupada B_η.
   b. Buscar en la canasta B_η al nodo mínimo.
      Recorrer secuencialmente la canasta B_η hasta encontrar al nodo cuya etiqueta sea la mínima, sea i tal nodo.
   c. Actualizar P_{max}: $P_{max} \leftarrow d(i)$.
   d. Actualizar L y H desde B_η hasta B_0.
   e. Reubica, aplicando la **Regla de Almacenamiento**, a los nodos de B_η.
2. Actualizar las etiquetas de los nodos adyacentes a i.

   Para todo j adyacente a i
   Si $d(j) > d(i) + a_{ij}$ **Entonces**
   $d(j) = d(i) + a_{ij}$;
   Reubicar al nodo j.

2.1 Tiempos por operación

Operaciones para una iteracion	Gasto
0. **Iniciación**	
o Sea $K = \log_2(nC)$	$\Theta(\log_2(nC))$
o Distribuir los n nodos en las Canastas	$O(n)$
$\star$ 1. **Buscar una Canasta, B no vacía**	$O(\log nC)^{\star}$
2. **Buscar en el Canasta B el nodo mínimo**	
o Suponiendo que todos los nodos están en la Canasta B	$O(n)^{\star}$
o En otro caso $(\star)$	$O(\log nC)^{\Delta}$
3. **Actualizar $\mathbf{P}_{max}$**	$\Theta(1)$
4. **Actualizar L[ ] y H[ ] desde B_0 hasta B_K**	$O(\log nC)^{\dagger}$
5. **Reubicar cada nodo**	$O(\log nC)^{\star}$

Notas

$\star$ Puede mejorar

$\star$ Puede ser constante

$\dagger$ Depende del paso 1

Δ Después de la segunda iteración

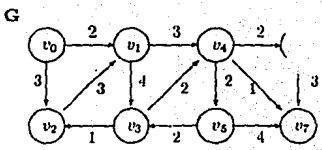


Figura 68: Red G.

3 Ejemplo Detallado

Considerese la Red G mostrada en la Figura 68, suponga que se desea encontrar la Ruta Más Corta entre v_0 y v_7.

Tenemos que: $|N| = n = 8 \quad |A| = m = 13 \quad C = 4.$

I. Inicio del Algoritmo

$nC = 32 \implies K = \lceil \log_2 nC \rceil = 6$ { 32 se representa con 6 bits }

$\implies$ se usarán (K+1) = 7 canastas.

Sea $P_{max} = 0$, las distancias (etiquetas) son:

$d(s) = d(v_0) = 0; \qquad d(v_j) = nC = 32, \quad j = 1, 2, \ldots, 7$

A la red G la representamos como una lista de Adyacencias (ver Figura 69) y consideraremos, sólo para ilustrar, a las distancias como un arreglo:

D :

0	32	32	32	32	32	32	32
0	1	2	3	4	5	6	7

Establecemos los rangos para las canastas:

[L , H]	Rango	[L , H]	Rango
[00, 00]	$R(0) = P_{max} = 000000$;	[08, 15]	$R(4) = 001000 \cdots 001111$;
[01, 01]	$R(1) = 000001$;	[16, 31]	$R(5) = 010000 \cdots 011111$;
[02, 03]	$R(2) = 000010 \cdots 000011$;	[32, 63]	$R(6) = 100000 \cdots 111111$;
[04, 07]	$R(3) = 000100 \cdots 000111$.		

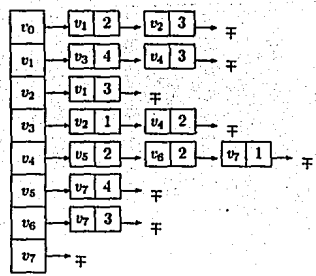
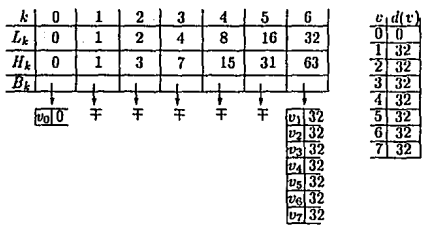


Figura 69: Lista de Adyacecias de G

Para facilitar la visualización del algoritmo, en la representación gráfica del R-Heap, añadimos a los nodos sus respectivas etiquetas. Distribuimos los n-nodos en las K-canastas:



II. Mientras Existan Canastas No Vacías.

Iteración 1.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_0.

El nodo de mínima distancia en B_0 es: v_0.

$P_{max} \longleftarrow d(v_0) = 0$.

Actualizar **L** y **H** de B_0 a B_0.

Reubicar los nodos de B_0.

2. Actualizar las etiquetas de los nodos adyacentes a v_0 y reubicarlos.

Los vecinos de v_0 son: v_1 y v_2.

$$d(v_1) = \min\{d(v_1), d(v_0) + a_{01}\} = \min\{32, 0 + 2\} = 2;$$

$$d(v_2) = \min\{d(v_2), d(v_0) + a_{02}\} = \min\{32, 0 + 3\} = 3.$$

Reubicar a los vecinos de v_0 :

v_1 va a la canasta B_2; v_2 va a la canasta B_2.

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	0	1	2	4	8	16	32
H_k	0	1	3	7	15	31	63
B_k			v_1 2 v_2 3				v_3 32 v_4 32 v_5 32 v_6 32 v_7 32

v_i	$d(v)$
0	00
1	02
2	03
3	32
4	32
5	32
6	32
7	32

Iteración 2.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_2.

El nodo de mínima distancia en B_2 es: v_1.

P_{max} —— $d(v_1) = 2$.

Actualizar L y H de B_0 a B_2.

[L, H] Rangos

[02. 02] $R(0) = P_{max} = 000010$;

[03. 03] $R(1) = 000011$: [− , −] $R(2) = \emptyset$.

Reubicar los nodos de B_2. $\Longrightarrow$ v_2 va a la canasta B_1.

2. Actualizar las etiquetas de los nodos adyacentes a v_1 y reubicarlos.

Los vecinos de v_1 son: v_3 y v_4.

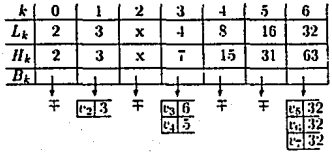
$$d(v_3) = \min\{d(v_3), d(v_1) + a_{13}\} = \min\{32, 2 + 4\} = 6;$$

$$d(v_4) = \min\{d(v_4), d(v_1) + a_{14}\} = \min\{32, 2 + 3\} = 5.$$

Reubicar a los vecinos de v_1 :

v_3 va a la canasta B_3; v_4 va a la canasta B_3.

Las canastas quedan estructuradas de la siguiente manera:



Iteración 3.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_1.

El nodo de mínima distancia en B_1 es: v_2.

P_{mar} — $d(v_2) = 3$.

Actualizar L y H de B_0 a B_1.

[L. H] Rangos

[03. 03] $R(0) = P_{mar} = 000011$: [- . -] $R(1) = \emptyset$.

Reubicar los nodos de B_1. { no hay. ha quedado vacío }

3. Actualizar las etiquetas de los nodos adyacentes a v_2 y reubicarlos.

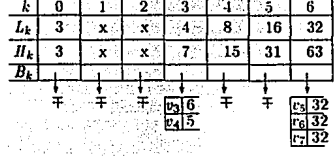
Los vecinos de v_2 son: v_1.

$d(v_1) = \min\{d(v_1), d(v_2) + a_{21}\} = \min\{2, 3 + 3\} = 6.$

Reubicar a los vecinos de v_2 :

{ No se modificó la etiqueta, entonces no se realiza }

Las canastas quedan estructuradas de la siguiente manera:



Iteración 4.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_3.

El nodo de mínima distancia en B_3 es: v_4.

$P_{max} \longrightarrow d(v_4) = 5$.

Actualizar **L** y **H** de B_0 a B_3.

[L, H] Rangos

[05, 05] $R(0) = P_{max} = 000101$ [− , −] $R(1) = \emptyset$

[06, 07] $R(2) = 000110 \cdots 000111$ [− , −] $R(3) = \emptyset$

Reubicar los nodos de B_3. $\Longrightarrow$ B_3 va a la canasta v_2

2. Actualizar las etiquetas de los nodos adyacentes a v_4 y reubicarlos.

Los vecinos de v_4 son: v_5, v_6 y v_7.

$d(v_5) = \min\{d(v_5), d(v_4) + a_{13}\} = \min\{32, 5 + 2\} = 7$:

$d(v_6) = \min\{d(v_6), d(v_4) + a_{13}\} = \min\{32, 5 + 2\} = 7$:

$d(v_7) = \min\{d(v_7), d(v_4) + a_{14}\} = \min\{32, 5 + 1\} = 6$.

Reubicar a los vecinos de v_4 :

v_5 va a la canasta B_2: v_6 va a la canasta B_2:

v_7 va a la canasta B_2.

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	5	x	6	x	8	16	32
H_k	5	x	7	x	15	31	63
B_k	↓	↓	↓	↓	↓	↓	↓
	∓	∓	v_3 6	∓	∓	∓	∓
			v_5 7				
			v_6 7				
			v_7 6				

v_i	$d(v_i)$
0	0
1	2
2	3
3	6
4	5
5	7
6	7
7	6

Iteración 5.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_2.

El nodo de mínima distancia en B_2 es: v_3.

$P_{max} \longrightarrow d(v_3) = 6$.

Actualizar **L** y **H** de B_0 a B_2.

[L, H] Rangos

[06, 06] $R(0) = P_{max} = 000110$;

[07, 07] $R(1) = 000111$; [− , −] $R(2) = \emptyset$.

Reubicar los nodos de B_2.

v_7 va a la canasta B_0: v_5 va a la canasta B_1:

v_6 va a la canasta B_1.

2. Actualizar las etiquetas de los nodos adyacentes a v_3 *y reubicarlos.*

Los vecinos de v_3 son: v_2 y v_4.

$d(v_2) = \min\{d(v_2), d(v_3) + a_{32}\} = \min\{3, 5 + 1\} = 3;$

$d(v_4) = \min\{d(v_4), d(v_3) + a_{34}\} = \min\{5, 6 + 2\} = 5.$

Reubicar a los vecinos de v_3 : { no se reubican, pues no cambiaron }

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	6	$\overline{7}$	x	x	8	16	32
H_k	6	$\overline{7}$	x	x	15	31	63
B_k	↓	↓	↓	↓	↓	↓	↓
	v_7 6	v_5 $\overline{7}$	∓	∓	∓	∓	∓
		v_5 $\overline{7}$					

v_i	$d(v)$
0	0
1	2
2	3
3	6
4	5
5	$\overline{7}$
6	$\overline{7}$
7	6

Iteración 6.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_0.

El nodo de mínima distancia en B_0 es: v_7.

$P_{max} \longleftarrow d(v_7) = 6.$

Actualizar **L** y **H** de B_0 a B_0.

Reubicar los nodos de B_2. { no hay más nodos en B_0 }

3. Actualizar las etiquetas de los nodos adyacentes a v_7 *y reubicarlos.*

{ v_7 no tiene vecinos }

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	6	7	x	x	8	16	32
H_k	6	7	x	x	15	31	63
B_k	↓	↓	↓	↓	↓	↓	↓
	∓	v_5 $\overline{7}$	∓	∓	∓	∓	∓
		v_5 $\overline{7}$					

v_i	$d(v)$
0	0
1	2
2	3
3	6
4	5
5	7
6	7
7	6

Iteración 7.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_1.

El nodo de mínima distancia en B_1 es: v_5.

$P_{max} \longrightarrow d(v_5) = 7$.

Actualizar **L** y **H** de B_0 a B_1.

[L, H] Rangos

[07.07] $R(0) = P_{max} = 000111$ [- , -] $R(1) = \emptyset$

Reubicar los nodos de B_1. $\Longrightarrow$ v_6 va a la canasta B_0

2. Actualizar las etiquetas de los nodos adyacentes a v_5 y reubicarlos.

Los vecinos de v_5 son: v_7.

$d(v_7) = \min\{ d(v_7), d(v_5) + a_{67} \} = \min\{ 6, 7 + 4 \} = 6$.

Reubicar a los vecinos de v_1: { no cambió, entonces no se reubica }

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	7	x	x	x	8	16	32
H_k	7	x	x	x	15	31	63
B_k	↓	↓	↓	↓	↓	↓	↓
v_0 7	∓	∓	∓	∓	∓	∓	∓

v_i	$d(v_i)$
0	0
1	2
2	3
3	6
4	5
5	7
6	7
7	6

Iteración 8.

1. Busca al nodo cuya etiqueta sea la menor.

La primera canasta *no vacía* es: B_0.

El nodo de mínima distancia en B_0 es: v_6.

$P_{max} \longrightarrow d(v_6) = 7$.

Actualizar **L** y **H** de B_0 a B_0.

2. Actualizar las etiquetas de los nodos adyacentes a v_6 y reubicarlos.

Los vecinos de v_6 son: v_7.

$d(v_7) = \min\{ d(v_7), d(v_6) + a_{67} \} = \min\{ 6, 7 + 3 \} = 6$.

Las canastas quedan estructuradas de la siguiente manera:

k	0	1	2	3	4	5	6
L_k	7	x	x	x	8	16	32
H_k	7	x	x	x	15	31	63
B_k	↓	↓	↓	↓	↓	↓	↓
	∓	∓	∓	∓	∓	∓	∓

v	$d(v)$
0	0
1	2
2	3
3	6
4	5
5	7
6	7
7	6

4 Análisis de Adaptabilidad del Radix Heaps

Lo que nos ocupa ahora es investigar si esta implantación resulta ser adaptiva o descubrir si el comportamiento del R-Heap e igual para todo tipo de red. Iniciaremos un análisis de Adaptabilidad con base en dos ideas elementales, que podrían darnos un panorama del comportamiento del DIJKSTRA.

Idea 1. Nuestra primera sospecha, intuitiva, se basa en creer que el desempeño computacional podría variar si el valor de C se modifica.

En efecto, el desempeño computacional del algoritmo cambia si C se ve alterado para una misma red, pero consideramos que C no es un parámetro suficientemente *fuerte* para determinar la adaptabilidad del algoritmo. La razón principal es que C puede representarse en diferentes unidades, lo cual no implica que el ejemplar ha cambiado.

Idea 2. Otra opción es *observar* el *trayecto* de cualquier nodo, diferente del origen, durante la ejecución del algoritmo.

Observamos que:

a) El vértice v, busca secuencialmente la canasta donde tiene que reubicarse, y de acuerdo con la *Observación 4* de Ahuja, Magnanti y Orlin [2], cada vez que un vértice se mueve, lo hace hacía una canasta con índice cada vez menor. Entonces, si el vértice $v \neq s$, viajará de la canasta B_K a B_0, ya que su primera etiqueta temporal lo ubica en B_K, después irá disminuyendo su etiqueta hasta convertirse en el mínimo.

b) Por otro lado, se busca secuencialmente una canasta no vacía, desde la primera canasta hacia otra con índice cada vez mayor, entonces se viajará de la canasta B_0 a B_K.

Estas observaciones dan pauta para pensar que, para algunos ejemplares, el vértice v *no* necesariamente debe recorrer las K-canastas, lo cual *disminuiría* el desempeño computacional del algoritmo.

Justificación de la Idea 2.

Sea $v \in N \setminus \{s\}$, esto es $v \neq s$. Supongase que en algún instante del algoritmo v se encuentra en la canasta B_j y v es el nodo de mínima etiqueta, entonces

- el vértice v tuvo que recorrer $(K - j)$ canastas para llegar a j; y
- para encontrar una canasta B_B no vacía hubo que recorrer j canastas.

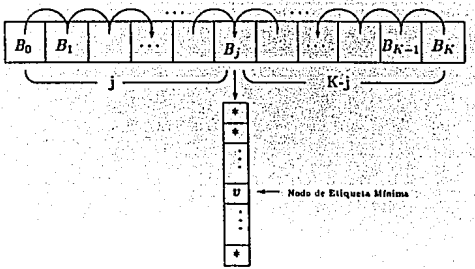


Figura 70: Ilustración del recorrido de un vértice.

La Figura 70 nos ilustra gráficamente el recorrido de un nodo. Tenemos, entonces que para el vértice v hubo que recorrer K canastas hasta encontrarlo como nodo mínimo; pero esto sucede $\forall v \in N \setminus \{s\}$, $(v \neq s)$. Por tanto, no importa como sea la red, el desempeño computacional del algoritmo siempre va ser: $\Theta(m + n \cdot \log_2(nC))$. Podemos concluir entonces que el algoritmo DIJKSTRA utilizando la versión descrita del R-Heap [2] no es adaptivo.

Lo que en realidad nos preocupa ahora es saber si esta implantación podría ser adaptiva, o si existe la posibilidad de que esta pueda ser modificada para que el algoritmo resulte ser adaptivo. Nótese que:

1. Modificar **L** y **H** gasta tiempo $O(\log_2 nC)$.

   ⋆ La *Observación 3* implica actualizar **L** y **H** sólo en las primeras B_* canastas.

2. Detectamos que las *búsquedas secuenciales*, utilizadas en los procesos de **Almacenamiento** y **BuscaMin**, hacen que el algoritmo sea lento.

Actualizando **L** y **H** sólo en las primeras B_*-canastas realizaremos un Análisis de Adaptabilidad para los procesos **Almacenamiento** y **BuscaMin** utilizando otro tipo de búsquedas.

Modificación al Proceso de Almacenamiento

Para la **Regla de Almacenamiento**, observamos que:

▷ v se almacena en B_j siempre que $L_j \leq d(v) \leq H_j$ $(\leq L_{j+1})$

$\Longrightarrow$ $L_0 \leq L_1 \leq L_2 \ldots \leq L_j \leq L_{j+1} \ldots \leq L_K$

$\Longrightarrow$ Los valores de los rangos **L** forman una *sucesión creciente*.

$\Longrightarrow$ Se puede realizar una búsqueda más sofisticada sobre los *indices* de los rangos **L** para reubicar a v. La *búsqueda binaria* sobre n elementos tiene desempeño computacional $O(\log_2 n)$ y la *Búsqueda Exponencial* [23] sobre n elementos tiene desempeño $O(\log_2 n)$.

La *distancia* que recorre cada nodo v para reubicarse resulta ser fundamental para el cálculo del desempeño computacional del algoritmo. El Análisis de Adaptabilidad que a continuación se presenta, está basado en tal distancia, además, usaremos una *Búsqueda Exponencial* [23], de B_K a B_0. Informalmente diremos que la Adaptabilidad del DIJSKTRA utilizando el R-Heaps dependerá de los *brinquitos* que dé cada vértice para reubicarse en las canastas.

Justificación del Análisis de Adaptabilidad

Sea $v \in N \setminus \{s\}$. Suponga que, en algún instante del algoritmo B_η es la primera canasta no vacía y en ella se encuentra v, que en ese instante resulta ser el nodo con mínima etiqueta. Se requiere reubicar a cada nodo $w \in B_\eta$ entre las canastas B_η y B_0. Suponga, además, que el lugar de la canasta que le toca al vértice ω es la posición j.

Sabemos que: $L_0 \leq L_1 \leq L_2 \ldots \leq L_j \leq L_{j+1} \ldots \leq L_{K-1} \leq L_K$ y $H_0 \leq H_1 \leq H_2 \ldots \leq H_j \leq H_{j+1} \ldots \leq H_{K-1} \leq H_K$, además, ω se almacena en el B_j siempre que: $L_j \leq d(w) \leq H_j$ $(\leq L_{j+1})$. Nótese que, si ω va a la canasta B_j entonces, habría que hacer un *salto* de longitud $(\eta - j)$. Realizamos una Búsqueda Exponencial [23] de L_η a L_0 para reubicar a ω. Al hacer la búsqueda sucede que:

a. Se encuentra la ubicación exacta para ω, entonces se finaliza este paso.

b. No se localiza la ubicación exacta para ω, entonces se procede a hacer una búsqueda binaria entre:

Salto Anterior	y	*Salto Actual*
(Porque le Falto)		(Porque se Paso)
J_A		J_D

Suponga que para llegar a J_D se realizaron p-pasos (saltos).

La Figura 71 muestra las distancias recorridas por un nodo para su reubicación.

Nótese, entonces, que:

▷ $distancia\,(J_A, \eta) = \eta - J_A$ y para llegar a J_A se realizaron $(p-1)$ saltos

$\Longrightarrow J_A = \eta - 2^{(p-1)}$.

$\Longrightarrow distancia\,(J_A, \eta) = \eta - (\eta - 2^{(p-1)}) = 2^{(p-1)}$.

▷ $distancia\,(J_D, \eta) = \eta - J_D$ y para llegar a J_D se realizaron (p) saltos

$\Longrightarrow J_D = \eta - 2^{(p)} \Longrightarrow distancia\,(J_D, \eta) = \eta - (\eta - 2^p) = 2^p$.

▷ $distancia\,(J_D, J_A) = J_A - J_D = (\eta - 2^{(p-1)}) - (\eta - 2^p)$

$= 2^p - 2^{(p-1)} = 2^{(p-1)}(2-1) = 2^{(p-1)}$.

$\Longrightarrow distancia\,(J_D, J_A) = 2^{(p-1)}$.

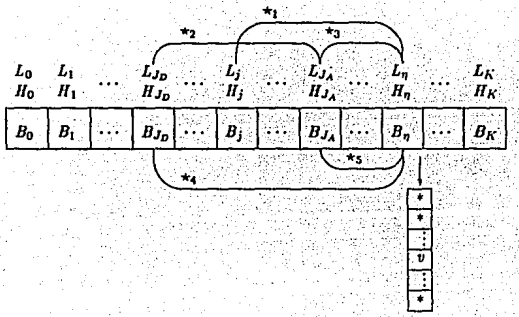
▷ $distancia\,(J, \eta) = \eta - J$.

De esta forma, tenemos que:

$distancia\,(J_D, \eta) > distancia\,(J, \eta) > distancia\,(J_A, \eta)$

$\Longrightarrow \quad 2^p > (\eta - J) > 2^{(p-1)}$

$\Longrightarrow \quad p > \log_2(\eta - J) > (p-1)$.



donde

$\star_1$: $\eta - J$
($L_{J_A} \longmapsto L_\eta$).

$\star_2$: $J_A - J_D$
($L_{J_D} \longmapsto L_{J_A}$).

$\star_3$: $\eta - J_A$
($L_{J_A} \longmapsto L_\eta$).

$\star_4$: p saltos, $\eta - J_D$
($B_{J_D} \longmapsto B_\eta$).

$\star_5$: $p-1$ saltos, $\eta - J_A$
($B_{J_A} \longmapsto B_\eta$).

Figura 71: Recorrido de un nodo para su reubicación

Pero esto es para un sôlo vértice. tomemoslo para todo vértice $n \in N$.

Sea $d(i) = (\eta - J) \ \forall \ i \in N$. entonces:

$$\sum_{i \in N} \log_2 (d(i)) = \log_2 \prod_{i \in N} d(i)$$

$$= \frac{NumViajes}{NumViajes} \times \log_2 \left(\prod_{i \in N} d(i) \right)$$

$$= (NumViajes) \times \log_2 \left(\prod_{i \in N} d(i) \right)^{(1/NumViajes)}$$

$$\leq (NumViajes) \times \log_2 \left(\frac{\sum_{i \in N} d(i)}{NumViajes} \right)$$

Por tanto hemos encontrado una *Función No Decreciente* que *depende* sólo del tamaño y la complejidad del *Número de Viajes ("brinquitos")*, que realizan los nodos para reubicarse en las canastas.

Propiedades utilizadas

- La suma de logaritmos es *equivalente* al logaritmo del producto:

$$\sum_{i=1}^{n} \log x_i = \log \left(\prod_{i=1}^{n} x_i \right).$$

- $\log x^{1/n} = \frac{1}{n} \log x.$
- La media geométrica nunca es mayor que la media aritmética:

$$\left(\prod_{i=1}^{n} x_i \right)^{(1/n)} \leq \frac{(\sum_{i=1}^{n} x_i)}{n}.$$

E Adaptabilidad en el Problema de Ordenamiento

En este Apéndice se resumen los conceptos y resultados fundamentales sobre el análisis de Adaptabilidad en el problema de ORDENAMIENTO. Hemos tomado como material base los artículos *A Survey of Adaptive Sorting Algorithms* escrito por V. Estivill-Castro y D. Wood [8] y *Measures of Presortedness and Optimal Sorting Algorithms* de H. Mannila [22].

1 Introduccion

El Problema de Ordenamiento, como es bien sabido, consiste en organizar una secuencia de elementos en orden ascendente o descendente. Pero, ¿qué significa que exista adaptabilidad en el problema de ordenamiento?, o más específicamente: ¿qué significa que un algoritmo sea adaptivo?. Una respuesta intuitiva nos la da Mehlhorn [23].

> "Cuando un algoritmo de ordenamiento toma ventaja del orden existente en los datos de entrada, el tiempo que toma el algoritmo para ordenar es una función suavemente creciente que depende del tamaño de la secuencia y del desorden en ella. En este caso decimos que el algoritmo es **adaptivo**."

Una panorámica sobre la importancia de la adaptabilidad en el problema de ordenamiento nos la dan Estivill-Castro y D. Wood [8].

> "El diseño y análisis de algoritmos de ordenamiento adaptivos, ha hecho importantes contribuciones, tanto para la teoría como para la práctica. Desde el punto de vista teórico, las contribuciones son:
>
> - la descripción del desempeño computacional del algoritmo de ordenamiento no sólo depende del tamaño de un ejemplar del problema sino, también, del desorden que hay en el ejemplar;
> - el establecimiento de nuevas relaciones entre las medidas del desorden;
> - la introducción de nuevos algoritmos de ordenamiento que toman ventaja del orden existente en la secuencia de entrada;
> - La prueba de que varios de los nuevos algoritmos de ordenamiento son adaptivamente óptimos con respecto a múltiples medias del desorden.
>
> "Las principales contribuciones desde el punto de vista práctico son:
>
> - la demostración de que múltiples algoritmos generalmente en uso son adaptivos.

- el desarrollo de nuevos algoritmos, similares a los comúnmente usados que se ejecutan competitivamente sobre secuencias aleatorias y son significativamente más rápidos sobre secuencias casi ordenadas.

"Los algoritmos de ordenamiento adaptivos son atractivos, porque las secuencias casi ordenadas son muy comunes en la práctica."

2 Medidas del Desorden

" Identificar en algún sentido casos fáciles de un problema computacional y utilizar esa facilidad tiene *interés esencial.* En ordenamiento, tal *facilidad* puede ser identificada con un orden existente." [22].

El número de operaciones que un algoritmo de ordenamiento ejecuta, es definido como la **medida de eficiencia.** El número de comparaciones proporciona no sólo una estimación razonable del tiempo relativo requerido para todas las implantaciones, sino que también permite cotas mínimas bajo el modelo computacional de árboles de decision.

A continuación describimos algunas medidas para cuantificar el orden existente en una secuencia, las cuales "son usadas en el estudio de adaptabilidad"[8] y las propiedades generales que deben satisfacer, de acuerdo a H. Mannila [22] las medidas del desorden.

2.1 Medidas Naturales.

Sea $\mathbf{X} = \langle x_1, x_2, \cdots, x_n \rangle$ la secuencia de datos a ordenar. Para simplificar, supongamos que los elementos x_i son enteros distintos. Consideremos que el orden ascendente es el correcto, esto es, se desea ordenar a la secuencia $\mathbf{X}$ en forma ascendente.

Inversiones. Dada una secuencia $\mathbf{X}$, las **Inversiones(X)** representan el número de parejas que se encuentran en orden incorrecto en $\mathbf{X}$ o que están en posición invertida. Formalmente, se definen como: $Inv(\mathbf{X}) = |\{(i,j) / 1 \leq i < j \leq n \text{ y } x_i > x_j\}|$.

El valor de $Inv(\mathbf{X})$ indica cuántos cambios entre cada par de elementos son necesarios para ordenar $\mathbf{X}$. Se tiene que $0 \leq Inv(\mathbf{X}) \leq n(n-1)/2$ para toda secuencia $\mathbf{X}$, de donde se obtiene, más específicamente,

$$Inv(\mathbf{X}) = \begin{cases} 0 & \mathbf{X} \text{ está ordenada en forma ascencente;} \\ \frac{n(n-1)}{2} & \mathbf{X} \text{ está ordenada en forma descencente.} \end{cases}$$

Ejemplo 1. Sea $Y = \langle 3, 1, 9, 4, 2, 10, 5, 7, 6, 8 \rangle$. El elemento en la posición uno de Y es mayor que el elemento en la posición dos, entonces la pareja (1,2) representa una inversión de Y. El conjunto de inversiones o parejas invertidas, para Y es:
$\{(1,2), (1,5), (3,4), (3,5), (3,7), (3,8), (3,9), (3,10), (4,5), (6,7), (6,8), (6,9), (6,10), (8,9)\}$.

Así que el número de inversiones de Y es, $Inv(X) = 14$.

Ejemplo 2. Sea $W = (n+1, n+2, \ldots, 2n, 1, 2, \ldots, n)$. Esta secuencia tiene un número cuadrático de inversiones.

Corridas. Dada una secuencia X, el valor de Runs(X) representa el número de subsecuencias ascendentes de X y se define formalmente como:

$$Runs(X) = \| \{ i \,/\, 1 \leq i < n \text{ y } x_{i+1} < x_i \} \| + 1.$$

Las subsecuencias ascendentes de X, están formadas por elementos en posiciones contiguas de X. por ello se les llama corridas (runs), se tiene que:

$$Runs(X) = \begin{cases} 1 & X \text{ está ordenada en forma ascencente;} \\ n & X \text{ está ordenada en forma descencente.} \end{cases}$$

Cuando la secuencia se encuentra casi ordenada, el valor de $Runs(X)$ es un número pequeño, pero cuando existe desorden local el número de corridas es grande.

Ejemplo 3. Para la secuencia Y del Ejemplo 1, $Runs(Y) = 6$, pues podemos ver a Y dividida de la siguiente forma: $Y = \langle 3, \downarrow 1, 9, \downarrow 4, \downarrow 2, \downarrow 10, \downarrow 5, 7, \downarrow 6, 8 \rangle$.

Ejemplo 4. Para la secuencia W del Ejemplo 2, $Runs(W) = 1$.

las Dada una secuencia X, el valor de las(X) representa el tamanõ de la subsecuencia ascendente más larga de X, y se define formalmente como:

$$las(X) = \max \{ t \,/\, \exists\ i(1), i(2), \ldots, i(t) \text{ tales que } 1 \leq i(1) < \cdots < i(t) \leq n \text{ y } x_{i(1)} < \cdots < x_{i(t)} \}.$$

Como el valor de $las(X)$ representa la longitud más de una corrida ascendente, se tiene entonces que $1 \leq las(X) \leq n$.

Ejemplo 5. Para la secuencia Y del Ejemplo 1, $las(Y) = 2$.

Ejemplo 6. Para la secuencia W del Ejemplo 2, $las(W) = n$.

Rem Dada una secuencia $\mathbf{X}$, el valor de $rem(X)$ indica cuantos elementos tienen que ser removidos de la secuencia para dejar la lista ordenada. Se define formalente como:

$$rem(\mathbf{X}) = n - las(\mathbf{X})$$

Se tiene que: $rem(\mathbf{X}) = \begin{cases} 0 & \mathbf{X} \text{ está en orden ascencente;} \\ n-1 & \mathbf{X} \text{ está en orden descendente.} \end{cases}$

Ejemplo 7. Para la secuencia $\mathbf{Y}$ del Ejemplo 1, $rem(\mathbf{Y}) = 10 - 2 = 8$.

Ejemplo 8. Para la secuencia $\mathbf{W}$ del Ejemplo 2, $rem(\mathbf{W}) = 2n - n = n$.

Cambios Dada una secuencia $\mathbf{X}$, el valor de $Exc(X)$ indica el número más pequeño de cambios necesarios de elementos arbitrarios para ordenar la secuencia $\mathbf{X}$ en forma ascendente.

Se tiene que: $0 \leq Exc(\mathbf{X}) \leq Inv(\mathbf{X}) \ \forall \ \mathbf{X}$.

Ejemplo 9. Para la secuencia $\mathbf{Y}$ del Ejemplo 1, $Exc(\mathbf{Y}) = 8$.

Ejemplo 10. Sea $\mathbf{W}_1 = (n, 1, 2, \ldots, (n-1))$. El número de cambios para esta secuencia es, $Exc(\mathbf{X}) = n - 1$.

Mannila [22], comenta algunas observaciones sobre estas medidas:

> "Comparar estas medidas no es fácil. *Inv* y *Runs* son medidas muy bien conocidas y su comportamiento es facil de comprender: las inversiones, *Inv*, cuantifican el pre-orden global de una secuencia, mientras que la medida *Runs* cuantica preorden local. Al parecer el valor de *Rem* combina estas dos propiedades, mientras que *Exc* parece diferir drasticamente de las otras, a pesar de su definición natural. Estas diferencias muestran que el pre-orden puede ser medido en muchas formas. "

2.2 Otra Medidas.

Distancia. Dada una secuencia $\mathbf{X}$, se define $\mathbf{Dis(X)}$ como la distancia más grande determinada por una inversión o una pareja invertida.

Esto hace énfasis en que "...una inversión, para la cual los elementos están muy apartados el desorden es más significante, que una inversión en la cual los elementos estan muy cerca el uno del otro" [8].

Ejemplo 11. Para la secuencia **Y** del Ejemplo 1, $Dis = 7$, ya que la $inv\,(3, 10)$ tiene la distancia más grande entre las $Inv\,(\mathbf{X})$. Viendolo de otra manera: el número nueve, elemento de la posición 3, esta a siete posiciones del número ocho, elemento de la posición 10; y son los elementos más apartados de la secuencia **Y**.

Máxima Distancia. Dada una secuencia **X**, se define **Max(X)** como la máxima distancia que un elemento debe viajar hasta encontrar su posición correcta.

Esta medida considera que el desorden local no es tan importante como el global [8].

Ejemplo 12. Para la secuencia **Y** del Ejemplo 1, $Max\,(\mathbf{Y}) = 6$, pues el elemento de la posición 3, el número nueve, debe viajar seis lugares para llegar a su posición correcta que es la nueve.

SUS Dada una secuencia **X**, el valor de **SUS(X)** indica el mínimo número de subsecuencias ascendentes en las que podemos particionar **X**.

La medida *SUS* es una generalización natural de las corridas, su nombre proviene de *Shufflet Up-Sequences*.

Ejemplo 13. Para la secuencia **Y** del Ejemplo 1,

$$SUS\,(\mathbf{Y}) = |\{\langle 3, 9, 10\rangle; \langle 1, 4, 5, 7, 8\rangle; \langle 2, 6\rangle\}| = 3.$$

SMS Dada una secuencia **X**, el valor de $SMS\,(X)$ indica el mínimo número de subsecuencias (ascendentes o descendentes) en que podemos particionar la secuencia dada.

Esta medida del desorden, $SMS\,(X)$, es una generalización de la medida $SUS\,(X)$. *SMS* es abreviación de *Shuffled Monotone Subsequence*.

Ejemplo 14. Para la secuencia **Y** del Ejemplo 1, $SMS\,(\mathbf{Y}) = 3$.

Ejemplo 15. Sea $\mathbf{W'} = \langle 6, 5, 8, 7, 10, 9, 12, 11, 4, 3, 2\rangle$.

Para esta secuencia de datos $\mathbf{W'}$ se tiene que:

$$Runs\,(\mathbf{W'}) = 7.$$

$$SUS\,(\mathbf{W'}) = |\{\langle 6, 8, 10, 12\rangle; \langle 5, 7, 9, 11\rangle; \langle 4\rangle; \langle 3\rangle; \langle 2\rangle; \}| = 5.$$

$$SMS\,(\mathbf{W'}) = |\{\langle 6, 8, 10, 12\rangle; \langle 5, 7, 9, 11\rangle; \langle 4, 3, 2\rangle; \}| = 3.$$

Enc Dada una secuencia $\mathbf{X}$, el valor de $Enc(X)$ se define como el número de listas ordenadas construidas por MELSORT al aplicarlo a la secuencia dada. Esta medida del desorden fué propuesta por Skiena [37].

Osc La Medida $Osc(X)$ evalua, en algunos casos la *Oscilación* entre el elemento más grande y el más pequeño de la secuencia $\mathbf{X}$. Esta medida del desorden fué propuesta por Levcopoulos y Petersson [20].

Reg Esta medida es definida por Moffat y Peterson [32, 33]; y nos indican que cualquier algoritmo de ordenamiento *Reg*-óptimo es optimamente adaptivo con respecto a las medias $Inv, Dis, Max, Exc, Rem, Runs, SUS, SMS, EncyOsc$.

2.3 Propiedades Generales.

En esta sección se dan algunas de las condiciones generales que debe satisfacer toda medida del pre-orden.

Definición Una **Medida** m es una función real, $m : \mathbf{N}^{<\mathbf{N}} \longrightarrow \mathbf{N}$, donde $\mathbf{N}^{<\mathbf{N}}$ denota al conjunto de todas las secuencias finitas de enteros (distintos) [22].

Condiciones.

1. $m(\mathbf{X}) = 0$, si la secuenicia $\mathbf{X}$ está ordenada en forma ascendente.
2. Sean las secuencias $\mathbf{X} = (x_1, x_2, \cdots, x_n)$ y $\mathbf{Y} = (y_1, y_2, \cdots, y_n)$. Si $x_i < x_j$ si y sólo si $y_i < y_j$ para todo i y j, entonces, $m(\mathbf{X}) = m(\mathbf{Y})$.
3. Si $\mathbf{X}$ es una subsecuencia de $\mathbf{Y}$ entonces, $m(\mathbf{X}) \leq m(\mathbf{Y})$.
4. Dadas las secuencias $\mathbf{X}$ y $\mathbf{Y}$. Si $\mathbf{X} < \mathbf{Y}$, esto es, que cada elemento de $\mathbf{X}$ es más pequeño que cada elemento de $\mathbf{Y}$, entonces, $m(\mathbf{XY}) \leq m(\mathbf{XY}) + m(\mathbf{Y})$.
5. Sea $a \in \mathbb{N}$, entonces $m(\langle a \rangle \mathbf{X}) \leq |\mathbf{X}| + m(\mathbf{X})$.

Las primeras dos condiciones indican que una lista ordenada tiene desorden igual a cero y que el valor de la medida m depende únicamente del orden de sus argumentos. La siguientes tres, son condiciones más fuertes, de hecho, de acuerdo con H. Mannila [22], son necesarias para toda medida del desorden.

H. Mannila [22], da dos puntos de vista sobre el concepto de pre-orden o desorden:

a) El desorden es cuantificado por el número de operaciones de un tipo dado, el cual es necesario para ordenar la secuencia de entrada (aproximación concreta).

b) El desorden es cuantificado por la cantidad de información de la forma $x_i < x_j$, la cual es requerida para identificar la secuencia usando una forma dada de coleccionar la información (aproximación de información teórica).

Estos puntos de vista obligan a las medidas del desorden a satisfacer las últimas tres condiciones. El punto de vista dado por a), asume que el conjunto de operaciones permitidas es cerrado bajo las subsecuencias. Si podemos ordenar una 'super-secuencia' de $\mathbf{X}$ con un cierto número de operaciones, entonces las restricciones de tales operaciones para $\mathbf{X}$ ordenarán $\mathbf{X}$; si $\mathbf{X} < \mathbf{Y}$, entonces $\mathbf{XY}$ (la concatenación de $\mathbf{X}$ con $\mathbf{Y}$) podrá ser ordenada si primero ordenamos $\mathbf{X}$ y luego $\mathbf{Y}$, con un total de $m(\mathbf{X}) + m(\mathbf{Y})$ operaciones; finalmente ordenar $(a)\,\mathbf{X}$, puede hacerse ordenando $\mathbf{X}$ con $m(\mathbf{X})$ operaciones e insertando a en su lugar correspondiente en la secuencia, por lo cual a tiene $|\mathbf{X}|$ diferentes posibilidades de ubicarse en la secuencia. De acuerdo con b), que si tenemos suficiente información para identificar una super-secuencia de $\mathbf{X}$, entonces esa misma información identifica a $\mathbf{X}$; si $\mathbf{X} < \mathbf{Y}$, entonces $\mathbf{XY}$ es identificada por la identificación de $\mathbf{X}$ y $\mathbf{Y}$; y $(a)\,\mathbf{X}$, es identificada al identificar $\mathbf{X}$ y después localizar el lugar de a [22].

3 Algoritmos Optimos

La medida de eficiencia de una algoritmo de ordenamiento, es el núm,ero de comparaciones que ejecuta, el cual no sólo proporciona una estimación razonable del tiempo relativo requerido para todas las implantaciones, sino que también permite cotas mínimas para ser obtenidas bajo el modelo computacional de Arboles de decisión [8].

H. Mannila [22] nos proporciona las siguientes definiciones sobre algoritmos óptimos y cotas mínimas.

Sean $\mathbf{X}$ la secuencia a ordenar, m una medida del pre-orden y z un entero positivo. H. Mannila [22], define

$$\textbf{below}'(z,\mathbf{X},m) = \{\, Y \in \mathbf{N}^{<\mathbf{N}} \,/\, Y \text{ es una permutación de } \{1,\ldots,|\mathbf{X}|\} \; y \; Y(\leq) z \,\}$$

donde $m(\mathbf{X}) \leq z$ y

$$\textbf{below}(\mathbf{X},m) = \{\, Y \in \mathbf{N}^{<\mathbf{N}} \,/\, Y \text{ es una permutación de } \{1,\ldots,|\mathbf{X}|\} \; y \; m(Y) \leq m(\mathbf{X}) \,\}$$

El conjunto **below'**$(z, \mathbf{X}, m)$ representa las permutaciones de los elementos de $\mathbf{X}$ con no más desorden que z con respecto a m. Por su parte el conjunto **below**$(\mathbf{X},m)$ representa las permutaciones para x con no más desorden que el que ya se tenía en $\mathbf{X}$, con respecto a la medida m.

Consideramos a los algoritmos que tienen como entrada no sólo la secuencia $\mathbf{X}$ sino también una cota superior z sobre el valor de $m(\mathbf{X})$. Un algoritmo S que usa $T_S(\mathbf{X}, z)$ pasos sobre la secuencia de entrada $\mathbf{X}$ y z es **debilmente optimo con respecto a** m, si para alguna $c > 0$ y para toda $\mathbf{X}$ y z, se tiene que:

$$T_S(\mathbf{X}, z) \leq c \cdot \max\{|\mathbf{X}|, \log|\text{below}'(z, \mathbf{X}, m)|\}.$$

Sea m una medida del preorden, y S un algoritmo de ordenamiento que usa $T_S(\mathbf{X})$ pasos sobre la secuencia $\mathbf{X}$. Se dice que S es **m-óptimo** u óptimo con respecto a m, si para alguna $c > 0$ y para toda $\mathbf{X}$, se tiene que

$$T_S(\mathbf{X}) \leq c \cdot \max\{|\mathbf{X}|, \log|\text{below}(z, \mathbf{X})|\}.$$

De esta forma, se define que un algoritmo de ordenamiento es optimamente adaptivo con respecto a una medida del desorden si este toma un numero de comparaciones que es un factor constante de la Cota Mínima (Lower Bound).

3.1 Clasificación entre Medidas.

La medida *Reg*, nos induce a las siguientes preguntas: ¿son estas medidas diferentes? o ¿Cómo pueden relacionarse entre sí?, Estivill-Castro y Wood [8] nos indican que: "Desde un punto de vista algorítmico, si dos medidas M_1 y M_2 particionan el conjunto de permutaciones en exactamente las mismas clases de conjuntos **below**, entonces cualquier algoritmo que es M_1-óptimo es también M_2-óptimo y viceversa." Además, nos proporcionan las siguientes definiciones:

Definición. Sean M_1 y M_2 dos medidas del desorden. Decimos que:

a) M_1 es **algorítmicamente más fina que (algorithmically finer)** M_2, denotado por $M_1 \leq_{alg} M_2$, si y sólo si cualquier algoritmo M_1-óptimo es también M_2-óptimo

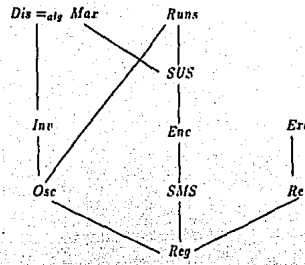


Figura 72: Orden parcial entre las *Medias del Desorden.*

b) M_1 y M_2 son **algoritmicamente equivalentes** si y sólo si $M_1 \leq_{alg} M_2$ y $M_2 \leq_{alg} M_1$. se denota como $M_1 =_{alg} M_2$.

La Figura 72 es dada por Estivill-Castro y D. Wood [8] para ilustrar el orden parcial entre las medias del desorden. con respecto a $\leq_{alg}$. la forma de "leer" el diagrama es de abajo hacia arriba: por ejemplo si un algoritmo es Inv-óptimo implica que es Dis-óptimo.

3.2 Ejemplos de Algoritmos Optimos

A continuación mostraremos algunos ejemplos de algoritmos óptimos dados por H. Mannila [22], quien nos dice que

> "Generalmente, las pruebas de que un algoritmo S es m-óptimo consisten en dos partes:
> 1) calcular el desmpeño computacional de S para secuencias Y y con $m(Y) \leq z$ y
> 2) estimar el tamaño del correspondiente conjunto below."

El algoritmo *Natural_MergeSort* [17] es un método de ordenamiento el cual inicia con las corridas existentes en la entrada y como una primera fase las mezcla por parejas, esta mezcla de parejas se continúa hasta que sólo existe una corrida.

Teorema 1 *El algoritmo Natural_MergeSort es óptimo con respecto al número de corridas.*

Demostración.

Sea X la secuencia de longitud n a ordenar y sea $t = Runs(X)$ el número de subcadenas ascendentes de X. Entonces el desmpeño computacional del algoritmo es proporcional a $n(1 + \log t)$, pues el número de fases-mezcla es acotado por $(1 + \log t)$ y cada fase toma tiempo lineal. Para mostrar que el algoritmo es óptimo, falta calcular el tamaño de el conjunto below(X, runs). Los siguientes resultados realizan tal cálculo.

Lema 2 *Sean $n, t \in N$. Si n y t son tales que $t \leq (n+1)/2$, entonces hay al menos $5^{-t}t^n$ permutaciones de $\{1, \ldots, n\}$ con exactamente t corridas.*

Corolario 3 *Para algunas $c, d > 0$ y para toda secuencia X se tiene que*

$$\log |\text{below}(X, runs)| \geq c \cdot n(\log t) - d \cdot t$$

siempre que $|X| = n$ y t es el número de corridas en X.

Demostración.

Si $t \leq (n+1)/2$ por el Lema 2 hay al menos $5^{-t}t^n$ permutaciones Y con Runs(Y) = Runs(X). Todas ellas pertenecen al conjunto below(X,runs), por lo tanto la afirmación se cumple. Si $(n+1)/2 < t \leq n$, entonces el conjunto below(X,runs) contiene al menos la mitad de las permutaciones de $\{1, \ldots, n\}$ H. Mannila [22] demuestra que para este caso también el resultado se cumple.

Finalmente, se tiene con estos resultados (Lema 2 y Corolario 3) que el Teorema 1 se sigue fácilmente de la definición de algoritmo in-óptimo.

Ordenamiento de Inserción Local.

Suponga que se tiene una estructura de datos para representar listas ordenadas, y que tal estructura es capaz de hacer inserciones en tiempo $O(1 + \log(d+1))$, donde d es la distancia del previo elemento insertado. Un algoritmo de ordenamiento que se implanta usando dicha estructura de datos es llamdo ORDENAMIENTO DE INSERCIÓN LOCAL (Local Insertion Sort). El tiempo de ejecución de este algoritmo es $\sum_j c \cdot (1 + \log(d_j + 1))$ donde d_j es la distancia desde el elemento anterior:

$$d_j = |\{ i / 1 \leq i < j \text{ y } (x_{j-1} < x_i < x_j \text{ o } x_j < x_i < x_{j+1}) \}|.$$

Teorema 4 *El algoritmo de* INSERCIÓN LOCAL *es óptimo con respecto al número de inversiones.*

Demostración

La eficiencia del algoritmo de INSERCIÓN LOCAL depende de la distancia entre inserciones sucesivas. En un ejemplar que contiene pocas inversiones, la mayoría de las inserciones ocurren cerca del final de la lista. En cada secuencia la distancia entre inserciones sucesivas debe ser pequeña, ya que es acotada por la mayor distancia desde el final de la lista de elementos a insertar y los elementos previos.

Si h_j es la cantidad que describe la distancia del elemento x_j desde el final de la lista, entonces se tiene que:

$d_j + 1 \leq \max\{h_j, h_{j-1}\} + 1 \leq h + j + h_{j+1} + 1 \leq (h_j + 1)(h_{j-1} + 1)$.

Así que. $\log(d_j + 1) \leq \log(h_j + 1) + \log(h_{j-1} + 1)$

y por tanto:

$$\sum_j c(1 + \log(d_j + 1)) \leq cn + c\left(\sum_j \log(d_j + 1)\right)$$

$$\leq cn + 2c\left(\sum_j \log(h_j + 1)\right) \leq cn + 2cn \log\left(\prod_j \log(h_j + 1)^{1/n}\right)$$

$$\leq cn + 2cn \log\left(\sum_j \log(h_j + 1)/n\right) \leq cn + 2cn \log(1 + Inv(X)/n).$$

Finalmente, tenemos que:

$$\log |\,\text{below}(X, Inv)\,| = \Theta(n \log(1 + Inv(X)/n))$$

lo cual demuestra que el algoritmo es óptimo con respecto a la medida del desorden denominada *inversiones*.

Teorema 5 *El algoritmo de ordenamiento de* Inserción Local *es óptimo con respecto al número de corridas, medida Runs.*

Teorema 6 *(H. Mannila [22], Teorema 8) El algoritmo de* Inserción Local *es óptimo con respecto a la medida Rem.*

Bibliografía

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Networks Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] R.K. AHUJA, T.L. MAGNANTI, and J. B. ORLIN. Some recent advances in network flows. *SIAM Review*, 33(2):175–219, June 1991.

[3] GRADY BOOCH. *Object Oriented Design. With Aplications.* Cummings Publishing Co., California,USA, 1991.

[4] G.B. DANTZING. On the shorest-route through a network. *Management Sci.*, 6:187–190, 1960.

[5] E. DIJKSTRA. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[6] BRUCE ECKEL. *Using C++.* Osborne, McGraw-Hill, USA, second edition edition, 1990.

[7] VLADIMIR ESTIVILL-CASTRO. *Sorting and Measures of Disorder.* PhD thesis, Faculty of Mathematics, University of Waterloo, February 1991.

[8] VLADIMIR ESTIVILL-CASTRO and DERICK WOOD. A survey of adptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1972.

[9] G.H. GONNET and R. BAEZA-YATES. *Handbook of Algorithms and Data Structures.* Addison-Wesley Publishing Co., USA, second edition edition, 1991.

[10] S.E. GOODMAN and S.T. HEDETNIEMI. *Introduction to the Design an Analysis of Algorithms.* International Students Edition. McGraw-Hill, USA, 1977.

[11] ADOLFO GUZMÁN-ARENAS. *Diseño y Construcción de aplicaciones con objetos.* IV Escuela Internacional de Invierno en Temas Selectos de la Computación. IIMAS, UNAM, Mérida, Yucatan, México, 1993.

[12] R.F. HILLE. *Data Abstraction and Program Development using Pascal.* Prentice Hall Co., Australia, 1988.

[13] J.A. HILLIER and P.D. WHITING. A method for finding the shorest toute through a road network. *Operations Res. Quart*, 11:37–40, 1960.

[14] T.C. HU. A descomposition algorithm for shorest path in networks. *Operations Res.*, 16(1):91–102, Jan-Feb 1968.

[15] MARTIN J.J. *Data Types and Data Structures.* Prentice Hall International, N.J., USA, 1986.

[16] J.H. KINGSTON. *Algorithms and Data Structures: Design, Correctness and Analysis.* Addison Wesley Co., Australia, 1990.

[17] D.E. KNUTH. *The Art of Computer Programming*, volume III: Sorting and Searching. Addison Wesley Co., USA, 1973.

[18] L.E. LAWLER. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, USA, 1976.

[19] L.E. LAWLER. Shorest-path and network flows algorithms. *Annales of Discrete Math.*, 4:251–263, 1979.

[20] C Levcopoulos and OLA PETERSON. Heapsort - adapted for presorted files. In *Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 382, pages 449–509. Springer-Verlang, 1989.

[21] UDI MANBER. *Introduction to Algorithms. A Creative Approach.* Addison-Wesley Publishing Co., USA, 1st edition, 1989.

[22] HEIKKI MANNILA. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computer*, 34:318–325, April 1985.

[23] K MEHLHORN. *Data Structures and Algorithms. Vol I Sorting and Searching.* Monographs on Theorical Computer Science. Springer-Verlag, Berlin, 1984.

[24] BERTRAN MEYER. *Object Oriented Software Construction.* Series in Computer Science. Prentice Hall International, USA, 1988.

[25] BERTRAN MEYER. *Eiffel: The Lenguage.* Version 2.2 of the Eiffel Environment. Interactive. Software Engineering Inc., USA, 1989.

[26] BERTRAN MEYER. *Eiffel: The Libraries.* Version 2.3 of the Eiffel Environment. Interactive. Software Engineering Inc., USA, 1990.

[27] BERTRAN MEYER. *Eiffel. The Language.* Prentice Hall International, USA, 1992.

[28] BERTRAN MEYER and JEAN-MARC NERSON. *Eiffel: The Libraries.* Version 2.3 of the Eiffel Environment. Interactive. Software Engineering Inc., USA, 1991.

[29] G. MILLS. A descomposition algorithm for the shorest-route problems. *Operations Res.*, 14:279–291, 1966.

[30] FREDMAN M.L. New bounds on the complexity of the shortest path problem. *Siam Journal Computing*, 5(5):83–89, 1976.

[31] FREDMAN M.L. and TARJAN R.E. Fibonacci heaps and their uses in improved network , optimization algorithms. *Journal of ACM*, 34(3):596–615, 1987.

[32] A. MOFFAT and OLA PETERSON. Historical searching and sorting. In *Second Annual International Simposium on Algorithms*, Lecture Notes in Computer Science. Springer-Verlang, 1991.

[33] A. MOFFAT and OLA PETERSON. A framework for adaptive sorting. In *3rd Scandinavian Workshop on Algorithms Theory*, Lecture Notes in Computer Science 382. Springer-Verlang, 1992.

[34] BROWN M.R. Implementation and analysis of binomial queues algorithms. *Journal of ACM*, 7(3):298–319, August 1978.

[35] OLA PETERSON. Adptive selection sorting. Technical report, Department of Computer Science, Lund University, October 1991.

[36] JUAN PRAWDA. *Metodos y Modelos de Investigacion de Operaciones*, volume I Modelo Deterministicos. Limusa, Mexico, 1988.

[37] S.S. SKIENA. Encroaching list as a mesure of preorden. *BIT*, 28(28):755–784, 1988.

[38] Sun Desktop SPARC. *DeskSet Reference Guide.* Reference Manuals. Sun Microsystems, Inc., USA, 1991.

[39] Sun Desktop SPARC. *Sun Station SLC.* Reference Manuals. Sun Microsystems, Inc., USA, 1991.

[40] Sun Desktop SPARC. *Sun System and Network. Manager's Guide.* Reference Manuals. Sun Microsystems, Inc., USA. 1991.

[41] Sun Desktop SPARC. *Sun System User's Guide.* Reference Manuals. Sun Microsystems, Inc., USA, 1991.

[42] M. SPIRA. A new algorithm for finding all shortest paths in a graph of positive arcs in averange time $o(n^2 \log^2 n)$. *SIAM J. Computing*, 2(2):28–32, 1973.

[43] R.E. TARJAN. Complexity of combinatorial algorithms. *SIAM Review*, 20(3):457–491, 1978.

[44] R.E. TARJAN. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):245–281, 1984.

[45] ROBERT ENDRE TARJAN. *Data Structure and Network Algorithms.* CBMS-NFS. Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, USA, fifth edition, 1988.

[46] HU T.C. *Combinatorial Algorithms.* Addison-Wesley Publishing Co., Philippines, 1982.

[47] J. VUILLEMIN. A data structure for manipulating priority queues. *C. of ACM*, 21(4):309–315, 1978.

[48] Nickaus Wirth. *Algorith + Data Structures = Programs.* Automatic Computation. Prentice Hall, USA, 1976.