Addendum

A.1 Introducción

La librería `SAL` presentada en la Tesis Doctoral titulada "*Caracterización de los modelos de búsqueda de un agente con descripciones generalizadas de los nodos origen y destino*" contenía un error en la implementación de varios algoritmos, puesto de manifiesto por uno de los miembros del Tribunal. A continuación, se presentan los resultados sobre los mismos dominios probados en la Tesis, con el código que resulta de corregir dicho error.

Asimismo, se propone un modelo matemático, para discutir la utilidad de dicha librería en la comparación entre diferentes algoritmos, con el objetivo de obtener conclusiones sobre su comportamiento, en especial, sobre los algoritmos de perímetro.

Por último, se extienden los resultados obtenidos teóricamente, con el uso de un código propio de investigación, a un caso real en condiciones reales, esto es, sacando el máximo partido de las propiedas intrínsecas del dominio de aplicación. En concreto, al caso de la implementación especializada de Richard E. Korf que implementa el algoritmo IDA* para la resolución de los 100 casos de su juego de ensayo.

A.2 Identificación y corrección del error

A continuación se presenta la naturaleza y alcance de un error encontrado por uno de los miembros del Tribunal. Se discute, asimismo, la corrección realizada y el efecto que dicho cambio ha tenido en las pruebas realizadas sobre los dominios propuestos en la

Tesis.

A.2.1 Naturaleza y alcance del error

La sugerencia de la posibilidad de un error en `SAL`, se debió a la observación de que algoritmos con ocupación lineal de memoria como el IDA* y el RBFS, no eran capaces de resolver todos los casos probados del N-"Puzle". Como el exceso de memoria ocupada por la instanciación de las plantillas en que consiste `SAL` no parecía un motivo razonable, la explicación más plausible era la de un error de gestión de memoria.

El error consistía, en pocas palabras, en el exceso de memoria que se consumía en hacer algunas operaciones intermedias y que no era nunca liberado. En concreto, la más grave de estas operaciones se llevaba a cabo en el método `descendants` (página 142 de la Tesis Doctoral): en este caso se creaba siempre un nodo ficticio por cada nuevo sucesor que era comparado con aquellos de la lista cerrada del nodo expandido. Si el nodo había sido previamente visitado, se borraba la instancia ficticia empleada en la detección del caso, y con ella, las estructuras a las que apuntaba —consúltese la figura 5.14, en la página 141 de la Tesis Doctoral. Sin embargo, si el nodo no había sido visitado previamente, se añadía el nuevo nodo generado, sin eliminar la instancia ficticia utilizada en la detección de este caso. Resulta obvio que, con ello, el *goteo de memoria* crecía exponencialmente.

La corrección del problema, consiste en generar copias de los nodos expandidos que siempre son eliminados. Sobre ellos, es posible ahora crear instancias ficticias con cualquier propósito (comprobación de nodos visitados previamente, colisión con el nodo final o un nodo de perímetro, copia de la solución, etc.) que siempre son eliminadas, evitando así el peligroso consumo de memoria que padecía `SAL`.

Este mecanismo funciona perfectamente para el dominio del N- "Puzle" puesto que en él, todos los nodos generados, son creados en memoria por primera vez. Esto es, el espacio de estados no *pre-existe* en memoria, sino que se genera bajo demanda. Otro caso bien distinto, es el dominio general de la búsqueda en grafos, puesto que en este caso, el grafo completo existe en memoria antes de iniciar la búsqueda. Por este motivo, la eliminación de cualquier instancia residente en memoria está protegida para no eliminar los contenidos reales del grafo. Sin embargo, eso significa que aquellas instan-

cias que apunten directamente a nodos del grafo, ¡no serán nunca eliminadas!, creando efectivamente un *goteo de memoria*. En concreto, éste es el caso de las instancias de `SALInfoNode` que, cuando son eliminadas, nunca eliminan las instancias `SALNode` a las que apuntan, puesto que ellas contienen posiciones reales del grafo que siempre deben preservarse.

Esta disparidad en el uso de la memoria es conveniente para la comparación de los algoritmos de búsqueda implementados en `SAL`, puesto que sirve para observar el rendimiento de los mismos algoritmos de búsqueda probados en el dominio precedente, en circunstancias bien distintas.

En cualquier caso, es preciso hacer notar que:

- La nueva versión de `SAL` es ahora más capaz que la anterior, resolviendo más casos que los anteriores.
- Que la resolución de los mismos casos probados con la versión anterior de `SAL` consume ahora un poco más de tiempo, puesto que es preciso realizar un número alto de operaciones extraordinarias para evitar el *goteo de memoria*.
- Que el resto de los parámetros (nodos generados, número de evaluaciones heurísticas, factor de ramificación heurístico, etc.) no se altera en absoluto, puesto que el cambio introducido no afecta a los algoritmos, sino al modo en el que gestionan la memoria, exclusivamente.

Por todo ello, se advierte que los contrastes de hipótesis llevados a cabo en las secciones 6.3 y 6.4 de la Tesis Doctoral (páginas 155–195), siguen siendo válidos, puesto que allí se hicieron sobre un subconjunto representativo de casos resueltos por cada algoritmo.

En la sección A.6, las tablas A.1– A.10 (páginas 30 a 46), muestran los resultados de los mismos algoritmos probados en la Tesis Doctoral, sobre las 50 primeras instancias del juego de ensayo de Richard E. Korf. Asimismo, las tablas A.11– A.18, muestran los resultados de los mismos algoritmos sobre todas las instancias del dominio general de búsqueda en grafos.

A.3 Consideraciones matemáticas

En esta sección se presenta un modelo matemático que sirve para predecir el comportamiento de los algoritmos de perímetro, relativos al número de nodos generados y el tiempo consumido, en función del perímetro empleado. Dicho estudio, servirá para poner de manifiesto la utilidad de `SAL`, y por qué sirve para hacer comparaciones entre diferentes algoritmos de un agente, una vez que se han considerado las precauciones necesarias.

La motivación de este estudio teórico se debe a la observación de uno de los miembros del Tribunal, por la que, aunque los algoritmos de búsqueda comparten tanto código como es posible, no puede garantizarse que las comparaciones sean necesariamente justas, puesto que la eficiencia de la implementación puede afectar los resultados. Siendo ésto cierto, se mantiene que la implementación de `SAL` crea diferencias no significativas, en concreto, que no perjudica a ningún algoritmo hasta el punto de convertirlo en peor que otro y, en el estudio de los algoritmos de perímetro, que no sanciona ciertas profundidades de perímetro en beneficio de otras. Por todo ello, se mantiene que las comparaciones realizadas entre los algoritmos con el uso de esta librería son válidas.

A continuación se justifica esta afirmación a propósito del uso de los algoritmos de perímetro como el BIDA*, y sus diferencias con su versión unidireccional, el algoritmo IDA*, en el dominio del N-"Puzle".

A.3.1 Comportamiento de los algoritmos de perímetro

Los algoritmos de perímetro en general, y el BIDA* en particular, se benefician del uso de una heurística mejor informada, $h_d(n)$:

$$h_d(n) = \min_{m \in P_d} \{h(n,m) + h^*(m,t)\} \tag{A.1}$$

donde $h(n,m)$ es la definición heurística original, empleada en la versión unidireccional del algoritmo de perímetro, tal y como se definió en la sección 2.4.4.9, en la página 36 de la Tesis. Por lo tanto, la conveniencia de su uso se debe al beneficio otorgado por el

uso de una heurística mejor informada, a cambio de la sobrecarga computacional en la evaluación de cada nodo para calcular el mínimo de la ecuación anterior.

Por lo tanto, cualquier estudio teórico sobre el comportamiento esperado de los algoritmos de perímetro debe comparar dichos valores. Se define, por lo tanto:

$$\xi(n, d, p_d) = T_u(n, d) - T_p(n, d, p_d) \tag{A.2}$$

como la diferencia del tiempo empleado por un algoritmo unidireccional de búsqueda, en la exploración del nodo n a profundidad d, y su versión de perímetro, con una profundidad de perímetro p_d. Téngase en cuenta que cualquier implementación real, realizará otras muchas operaciones que las que se discuten a continuación. Por lo tanto, T_u y T_p serán expresiones que dominan el tiempo real de ejecución para las implementaciones unidireccional y de perímetro, respectivamente.

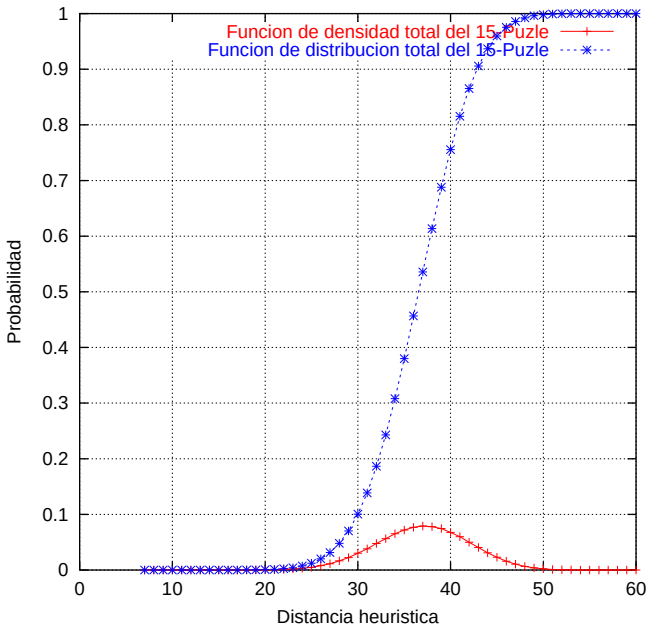


Figura A.1: Funciones de densidad y distribución total en el 15-"Puzle"

$T_u(n, d)$ se puede calcular a partir de la expresión del número de nodos generados, en el peor caso, por un algoritmo de búsqueda unidireccional admisible con una heurística que sigue la función de distribución total P, en un árbol con un factor de ramificación

b, y profundidad d, según la expresión:

$$N_1(b,d,P) = \sum_{i=1}^{d+1} b^i P(d-i+1)$$

expuesta en la sección 6.5 (página 195) de la Tesis, donde $P(x)$ es la probabilidad de que un nodo escogido al azar tenga una distancia menor o igual que x del nodo final, t, y que puede obtenerse por muestreo. Así, por ejemplo, la figura A.1 muestra las funciones de densidad total de la probabilidad $f_1(x) = p(h(n,t) = x)$, y la función de distribución total correspondiente a la probabilidad $F_1(x) = p(h(n,t) \leq x)$, obtenidas con el muestreo de mil millones de nodos.

Así, si t_h, t_e y t_c son los tiempos para la evaluación heurística de un nodo, su expansión, y la detección de colisión con el nodo final, respectivamente, se cumple:

$$T_u(n,d) = N_1(b,d,F_1)(t_h + t_c) + N_1(b,d-1,F_1)t_e \tag{A.3}$$

Por otra parte, en el caso de los algoritmos de perímetro, debe considerarse la sobrecarga computacional debida al uso del perímetro, y que debe descomponerse en dos partes diferentes: de una, el cálculo de la nueva función heurística, $h_{p_d}(n)$, según la ecuación (A.1); de la otra, el tiempo dedicado a comprobar si cada nuevo nodo generado es igual a alguno de los almacenados en el perímetro. Esto es, si ha habido alguna colisión con el perímetro P_{p_d}.

Además, debe tenerse en cuenta que el número de nodos generados o expandidos por el algoritmo de perímetro obedecerá a una expresión distinta de la anterior, puesto que al cambiar la definición heurística —más informada ahora—, cambiarán también las funciones de densidad y distribución total consideradas en el caso anterior y mostradas en la figura A.1. En cualquier caso, se define la función de densidad total, f_2, como la probabilidad $f_2(x,p_d) = p(h_{p_d}(n,t) = x)$. Por lo tanto, se define su función de distribución como $F_2(x,p_d) = (h_{p_d}(n,t) \leq x)$.

Obviamente, el tiempo dedicado a las expansiones por un algoritmo de perímetro será igual a:

$$N_1(b, d - p_d - 1, F_2)t_e$$

Para estimar el tiempo dedicado a las evaluaciones heurísticas y la comprobación de la colisión con el perímetro P_{p_d}, es preciso calcular el número medio de nodos de perímetro útiles en cada nivel i del árbol de búsqueda desarrollado en la búsqueda hacia delante, $|\bar{P_{p_d}}|^i$, y que será igual a:

$$\sum_{j=1}^{|P_{p_d}|} p(m_j) \times 1 + p(\tilde{m_j}) \times 0$$

donde $p(m_j)$ es la probabilidad de que el nodo de perímetro m_j, que pertenece al perímetro del nodo n, $P_{p_d}^n$, siga perteneciendo al perímetro de un sucesor suyo n_i, $P_{p_d}^{n_i}$; $p(\tilde{m_j})$ es la probabilidad del suceso contrario, es decir: $p(\tilde{m_j}) = 1 - p(m_j)$.

En el caso del N-"Puzle", la única forma de que un nodo m_j, que pertenece al perímetro de un nodo n, deje de hacerlo en uno de sus sucesores, n_i, es que se den simultáneamente las siguientes condiciones:

- El nodo n_i tiene una estimación heurística hasta el nodo m_j mayor que la de su padre:

$$h(n_i, m_j) - h(n, m_j) = 1$$

- El coste del nodo n, $f(n)$, es igual al umbral empleado en la búsqueda hacia delante, η:

$$g(n) + h(n, m_j) + k(m_j, t) = \eta$$

Sumando las dos expresiones anteriores, y operando, resulta:

$g(n) + h(n_i, m_j) + k(m_j, t) = 1 + \eta$	(sumando las expresiones anteriores)
$g(n) + h(n_i, m_j) = 1 + \eta - p_d$	(porque $k(m_j, t) = p_d$)
$h(n_i, m_j) = 1 + \eta - p_d - i$	(puesto que $c(n, n_i) = 1$, se tiene $g(n) = i$)

donde i es la profundidad a la que se encuentra el nodo n.

Asumiendo que la función de densidad total para el nodo m_j será como la del nodo terminal, t, la probabilidad de que un nodo m_j, que pertenecía al perímetro a profundidad p_d del nodo n, $P^n_{p_d}$, deje de hacerlo en el de su sucesor n_i, $P^{n_i}_{p_d}$ es:

$$p(\tilde{m_j}) = p(h(n_i, m_j) = 1 + \eta - p_d - i) = f_1(1 + \eta - p_d - i)$$

Como en los algoritmos de perímetro se ahorran necesariamente las últimas p_d iteraciones, se tiene que la profundidad, d, del árbol desarrollado en la búsqueda hacia delante es $d = \eta - p_d$, donde η es igual al coste de la última iteración.

Así, el número medio de nodos de perímetro útiles a profundidad $i + 1$ es:

$$|\bar{P_{p_d}}|^{i+1} = \sum_{j=1}^{|P_{p_d}|^i} (1 - f_1(1 + d - i))$$

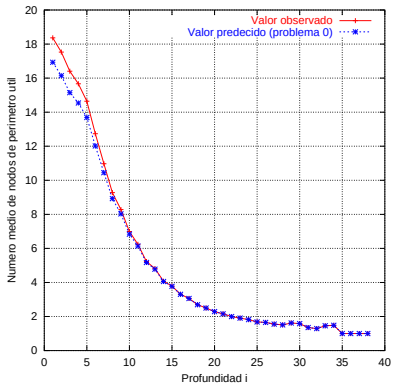
que es igual a:

$$|\bar{P_{p_d}}|^{i+1} = |P_{p_d}|^i (1 - f_1(1 + d - i)) \tag{A.4}$$

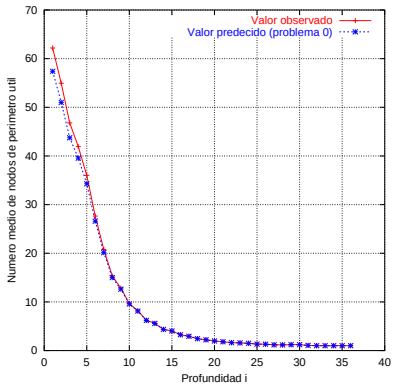
A modo de ejemplo, la figura A.2 muestra las diferencias entre el número medio de nodos de perímetro útil obtenido por muestreo de los problemas 0 y 1 (descritos en la tabla 6.1, en la página 155, de la Tesis) y el valor predecido por la ecuación anterior. Nótese que, tal y como predecía Giovanni Manzini (página 37 de la Tesis Doctoral), el número de nodos útiles de perímetro decrece con la profundidad del árbol de búsqueda desarrollado en la búsqueda hacia delante.

Por lo tanto, el número de evaluaciones heurísticas realizadas por un algoritmo de perímetro que emplea una función de distribución total $F_2(x, p_d)$, en el i-ésimo nivel del árbol de búsqueda, es igual a:

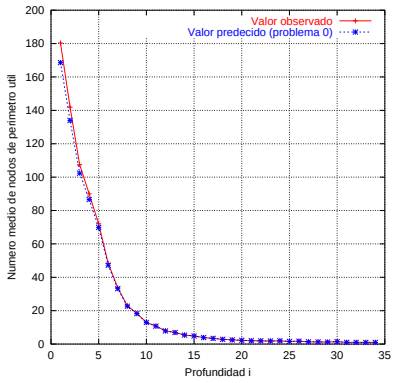
$$b^i F_2(1 + d - i)|P_d|^{i-1}(1 - f_1(2 + d - i))$$



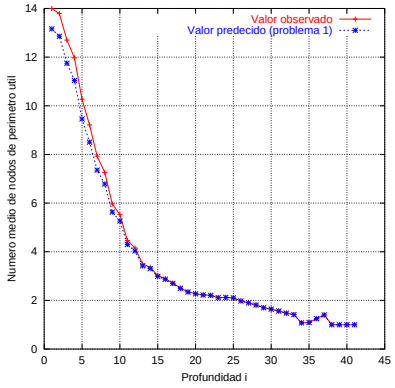
(a) Problema 0 a profundidad 4



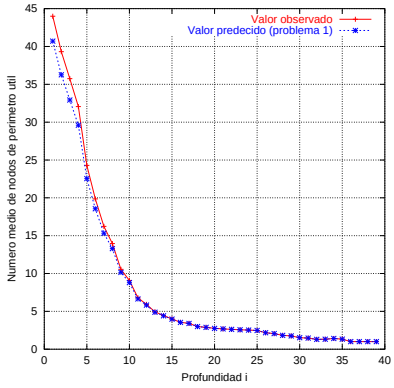
(b) Problema 0 a profundidad 6



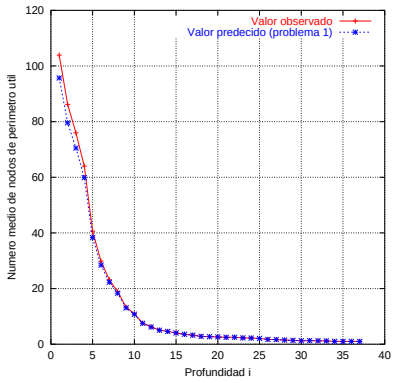
(c) Problema 0 a profundidad 8



(d) Problema 1 a profundidad 4



(e) Problema 1 a profundidad 6



(f) Problema 1 a profundidad 8

Figura A.2: Valores observados y predecidos del número medio de nodos de perímetro útil

donde d es la profundidad del árbol de búsqueda desarrollado en la búsqueda hacia delante, y que será p_d unidades menores que el desarrollado por el algoritmo unidireccional.

A propósito del tiempo consumido para la detección de colisiones con el perímetro, debe tenerse en cuenta que `SAL` emplea conjuntos representados como árboles binarios para almacenar los nodos de perímetro. Por ello, el tiempo para la búsqueda de un elemento en el perímetro tiene una complejidad $O(\log n)$, donde n es el número de nodos en el conjunto. Así, el número medio de accesos al perímetro para detectar colisiones en el nivel i-ésimo del árbol de búsqueda desarrollado en la búsqueda hacia delante por un algoritmo de perímetro, será igual a:

$$b^i F_2(1+d-i)\log(|P_{p_d}|^{i-1}(1-f_1(2+d-i)))$$

Por lo tanto, denominando:

$$N_2(b,d,P) = \sum_{i=1}^{d+1} b^i P(1+d-i)|P_{p_d}|^{i-1}(1-f_1(2+d-i))$$

$$N_3(b,d,P) = \sum_{i=1}^{d+1} b^i P(1+d-i)\log(|P_{p_d}|^{i-1}(1-f_1(2+d-i)))$$

se tiene que:

$$T_p(n,d,p_d) = N_2(b,d-p_d,F_2)t_h + N_1(b,d-p_d-1,F_2)t_e + N_3(b,d-p_d,F_2)t_c \quad \text{(A.5)}$$

Por lo tanto, una estrategia de perímetro, como el algoritmo BIDA*, con profundidad de perímetro p_d, consumirá menos tiempo en encontrar la solución óptima si:

$$\xi(n,d,p_d) = T_u(n,d) - T_p(n,d,p_d)$$

es estrictamente mayor que el tiempo dedicado a la generación del perímetro. Teniendo en cuenta que el perímetro se genera con la aplicación de un algoritmo de fuerza bruta, se considerará el factor de ramificación asintótico B, en vez del heurístico empleado

hasta ahora, b. Así, el número de nodos expandidos por un algoritmo de fuerza bruta a profundidad p_d será igual a:

$$\sum_{i=0}^{p_d-1} B^i = \frac{B^{p_d}-1}{B-1}$$

Por último, después de la generación del perímetro deben considerarse todas las inserciones de los nodos de perímetro generados a profundidad p_d en el perímetro. Como en el caso de las consultas al conjunto que implementa el perímetro para detectar colisiones, antes discutido, la complejidad del tiempo para la inserción de un nuevo nodo es $O(\log n)$, donde n es el número de nodos que hay actualmente en el perímetro. Por lo tanto, se tiene que el algoritmo de perímetro será más conveniente que su versión unidireccional si:

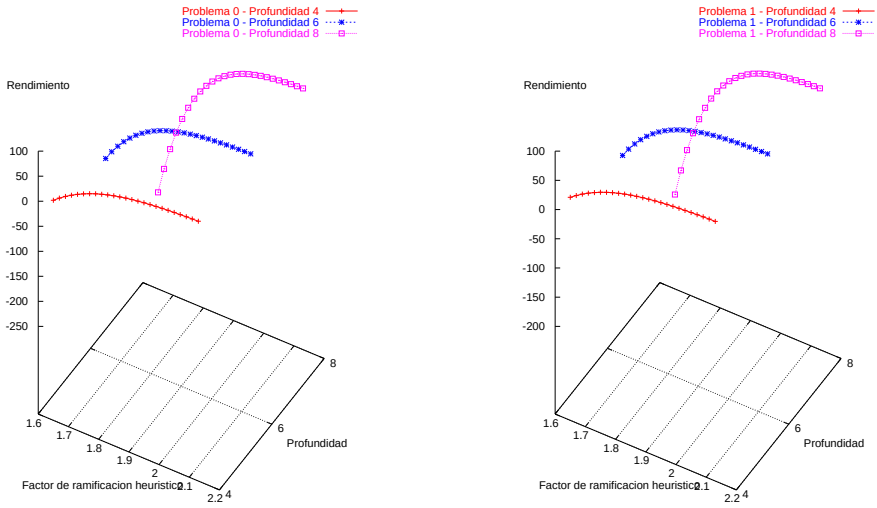
$$\xi(n,d,p_d) \geq \frac{B^{p_d}-1}{B-1}t_e + t_c \sum_{i=1}^{B^{p_d}} \log(i) = \frac{B^{p_d}-1}{B-1}t_e + t_c \log\left(B^{p_d}!\right) \qquad \text{(A.6)}$$

A.3.2 Ejemplos

Para los mismos problemas que los estudiados en la figura A.2 (problemas 0 y 1), se observó que los tiempos t_e, t_h y t_c son, aproximadamente, $1,53 \times 10^{-6}$, $1,25 \times 10^{-6}$ y $1,02 \times 10^{-6}$ segundos, respectivamente. Considerando, por otra parte, los factores de ramificación heurísticos mostrados en la tabla B.1 (página 230 de la Tesis Doctoral) $b = 2,05$ para el problema 0 y $b = 2,02$ para el problema 1, y usando el valor de ramificación asintótico, $B = 2,13$, derivado por Edelkamp y Richard Korf (tabla 3.1, página 84 de la Tesis Doctoral), el análisis propuesto muestra que:

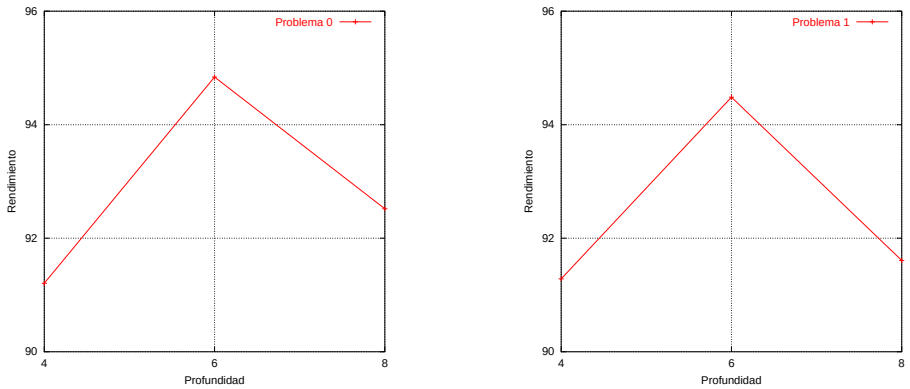
- La figura A.3(a) muestra el rendimiento teórico del algoritmo BIDA* sobre su versión unidireccional IDA* en el problema 0, para diferentes valores del factor de ramificación entre 1,65 y 2,13, y las tres profundidades de perímetro probadas en la Tesis Doctoral —4, 6 y 8. La figura A.3(b) muestra los mismos valores para el problema 1.

  Como se puede ver, la utilidad de los algoritmos de perímetro decrece cuanto menor es el factor de ramificación, creciendo para los valores mayores de b. Este



(a) Problema 0 (b) Problema 1

Figura A.3: Rendimiento teórico de los algoritmos de perímetro sobre su versión unidireccional



(a) Problema 0 (b) Problema 1

Figura A.4: Rendimiento teórico de los algoritmos de perímetro sobre su versión unidireccional (2)

resultado, completamente esperado, se debe al hecho de que para valores bajos del factor de ramificación heurístico, el número de nodos generados por cualquier implementación unidireccional decrece exponencialmente, haciendo sentir más el tiempo dedicado a la generación del perímetro y la evaluación heurística de cada nodo en los algoritmos de perímetro.

- Las figuras A.4(a) y A.4(b) muestran un corte transversal de las gráficas tridimensionales mostradas en las figuras A.3(a) y A.3(b), respectivamente, para los valores del factor de ramificación asintótico obtenidos muestralmente en cada problema y presentados anteriormente.

  Como se puede ver, la mejor profundidad de perímetro para ambos problemas es 6, **lo que coincide plenamente con los resultados presentados** en las tablas B.1, B.3, B.4 y B.5 (páginas 230, 233, 236 y 238 de la Tesis Doctoral), y las tablas A.1, A.3, A.4 y A.5 mostradas en este documento —páginas 30 y 33– 37.

A.3.3 Conclusiones

Son conclusiones de interés del análisis anterior las siguientes:

- Los algoritmos de perímetro se benefician, primero: del uso de una heurística, $h_{p_d}(\cdot)$, mejor informada que la empleada en su versión unidireccional, $h(\cdot)$; segundo: ahorrando las últimas p_d iteraciones de la búsqueda realizada hacia delante, precisamente las más costosas para la implementación unidireccional. A cambio, deben generar el perímetro P_{p_d}. El análisis anterior ha mostrado la relación entre estos tiempos.

- A propósito del uso de la heurística más informada, $h_{p_d}(\cdot)$, se hace notar que, al contrario de lo que cabría esperar intuitivamente, cuanto más informada esté la función heurística h_{p_d}, menor será el número de evaluaciones heurísticas que realizará la implementación de perímetro —consúltense los resultados obtenidos con `SAL`, en las tablas A.1, A.3, A.4 y A.5.

  Según Pearl, la heurística $h(n)$ estará más informada que $h_{p_d}(n)$, si $h_{p_d}(n) \geq h(n)$, para todo n, de donde se deriva fácilmente que $F_2(x, p_d) \leq F_1(x)$, para todo x,

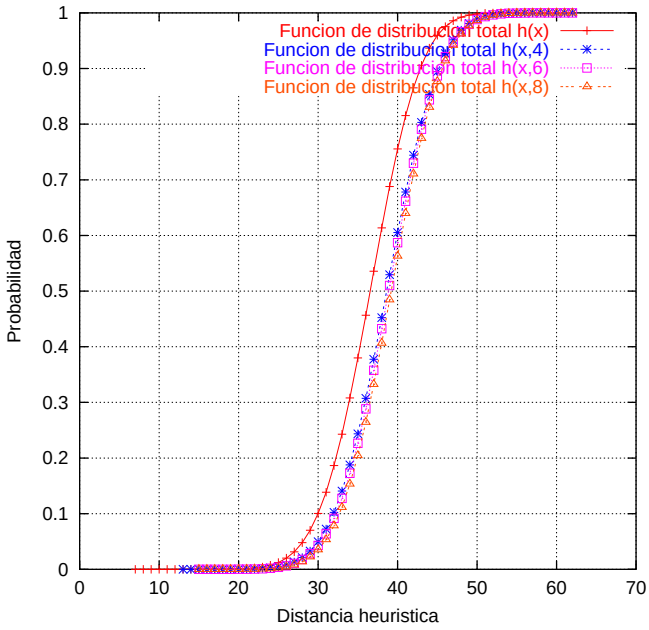


Figura A.5: Funciones de distribución total $F_1(x)$, $F_2(x,4)$, $F_2(x,6)$ y $F_2(x,8)$.

como puede observarse en la gráfica A.5, que muestra las funciones de distribución total $F_1(x)$, $F_2(x,4)$, $F_2(x,6)$ y $F_2(x,8)$, obtenidas muestralmente.

Puesto que las expresiones $F_2(x,p_d)$ y $F_1(x)$ dominan los tiempos obtenidos con N_1, N_2 y N_3, se observa que el número de nodos generados y el número de evaluaciones heurísticas tenderá a decrecer con la profundidad de perímetro.

- Si la estrategia de perímetro es más conveniente que la unidireccional en términos del tiempo consumido, se observa que la diferencia depende de la profundidad de perímetro p_d, habiendo un valor óptimo, que puede diferir, para cada profundidad del árbol de búsqueda desarrollado en la búsqueda hacia delante, d.

- Que la ganancia debida al uso de los algoritmos de perímetro, $\xi(n,d,p_d)$, será inevitablemente menor cuanto mayor sea la profundidad del árbol de búsqueda desarrollado en la búsqueda hacia delante. Por este motivo, es muy probable que esta técnica no sirva para resolver problemas del 24-"Puzle", aunque lo haga tan eficientemente en el dominio del 15-"Puzle" como se muestra en la Tesis, o como han indicado otros investigadores.

- Se llama la atención sobre el hecho de que el análisis presentado en este addendum, emplea la profundidad del árbol de búsqueda, d. Puesto que este parámetro no puede conocerse hasta que se ha resuelto el problema en cuestión, la línea futura mostrada en la página 214 de la Tesis Doctoral, en la que se propone la investigación de expresiones matemáticas que deriven automáticamente la profundidad de perímetro óptima sigue siendo válida, puesto que allí se hace referencia al uso, exclusivamente, de valores conocidos de antemano, antes de resolver el problema.

- En cualquier caso, la recomendación del uso de la estrategia de perímetro depende, tal y como se indica en la tabla 7.1 de la Tesis Doctoral (página 210) cuando la función de distribución total de la función heurística lo permite, siendo más aconsejables, cuanto menor sea la precisión de $h(\cdot)$, puesto que hay más posibilidades para que $h_{p_d}(\cdot)$ mejore las estimaciones originales y ahorre con ello un consumo considerable en la expansión y evaluación de nodos inútiles.

A.4 Un caso real

A pesar del estudio teórico presentado en la sección anterior, y los resultados obtenidos en esta Tesis, se debe tener en cuenta que todas las conclusiones presentadas tienen, aún, cierto carácter teórico, puesto que en la realidad se suele sacar el máximo partido posible de las características intrínsecas del dominio de aplicación, con el objeto de desarrollar algoritmos de búsqueda veloces. Por supuesto, considerando la especialización del dominio, mejorar los resultados de un algoritmo unidireccional con una versión de perímetro es más difícil, pero, a la luz del estudio mostrado en esta Tesis, debería ser posible.

A propuesta del mismo miembro del tribunal de esta Tesis, se ha desarrollado un código especializado para la resolución de todas las instancias del juego de ensayo diseñado por Richard E. Korf. El código mostrado en la sección A.5, saca partido de las mismas especializaciones ideadas por Richard E. Korf, explicadas en la sección 6.3.1 de la Tesis (página 159) y el apéndice A (página 223), además de otras, mostradas en la siguiente sección.

Es de especial importancia mencionar que el uso de tablas precomputadas ideadas por su autor, hace que el código del IDA* sea excepcionalmente rápido. En otros términos, los tiempos t_e y t_h son extraordinariamente bajos. Por este motivo, no pueden esperarse grandes ahorros de tiempo consumido con grandes perímetros. Por lo tanto, sólo se han hecho pruebas a profundidad 2 y 3.

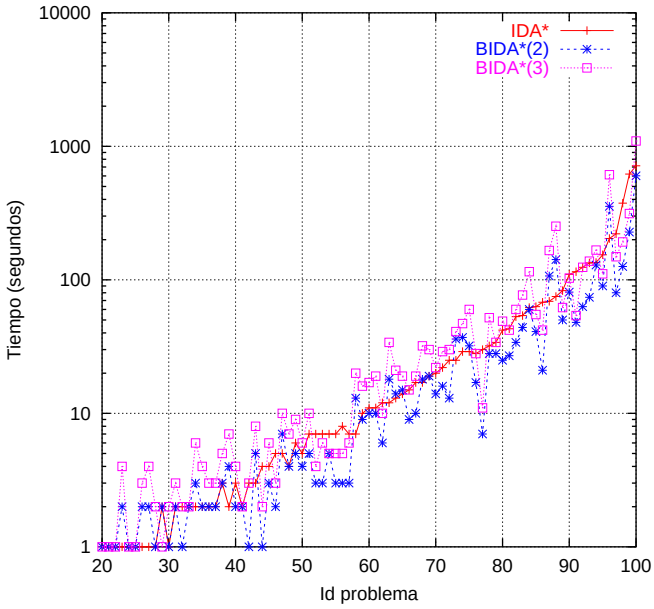


Figura A.6: Tiempos de implementaciones reales del IDA* y el BIDA* en el 15-"Puzle"

La figura A.6 muestra el tiempo consumido por el IDA*, el BIDA$^*_2$ y el BIDA$^*_3$, en escala logarítmica, para todos los problemas desde el 20 hasta el 100 —los anteriores no se muestran porque en muchos casos el tiempo es 0 segundos. Como puede verse, el algoritmo BIDA$^*_2$ es el más rápido en prácticamente todos los casos. Por el contrario, el algoritmo BIDA$^*_3$ es, con frecuencia, el más lento.

Por otra parte, la figura A.7 muestra el tiempo medio consumido por los mismos algoritmos, en escala logarítmica, para todos los problemas desde el 40 hasta el 100 —los anteriores no se muestran porque sus valores son demasiado bajos. Como puede verse, el algoritmo BIDA$^*_2$ es, efectivamente, el más rápido para la resolución de todo el juego de ensayo, ahorrando, en término medio, hasta 11,74 segundos. Esto es, el algoritmo BIDA$^*_2$ ahorra, en terminos absolutos, casi 20 minutos, sobre los 68 minutos que tarda

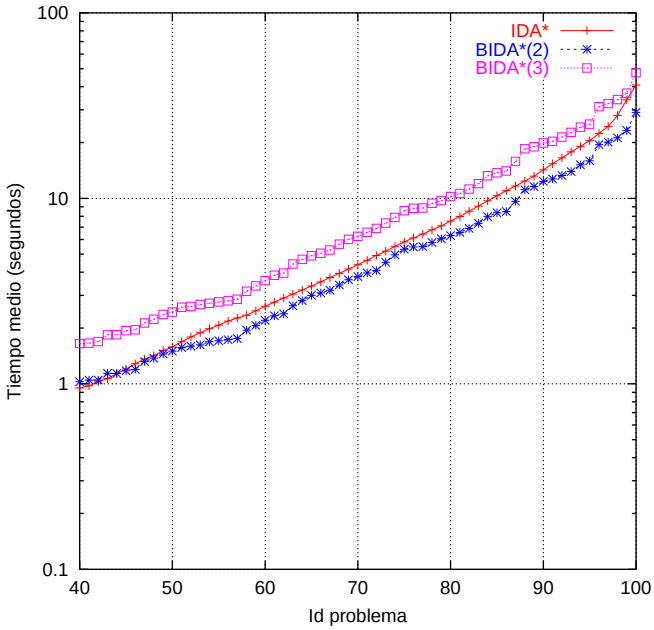


Figura A.7: Tiempos medios de implementaciones reales del IDA* y el BIDA* en el 15-"Puzle"

el algoritmo IDA*.

Por último, con el objeto de mostrar que el comportamiento real de los algoritmos de perímetro es como el esperado teóricamente, la figura A.8 muestra el número de nodos generados por cada algoritmo, en escala logarítmica, para los 100 problemas. Como puede verse, cualquiera de las dos versiones de perímetro genera muchos menos nodos que su versión unidireccional. El BIDA$^*_3$ es el que menos nodos genera, sin embargo, esta ventaja no es suficiente para mejorar los resultados del BIDA$^*_2$, puesto que, como ya se ha mencionado, los tiempos t_e y t_h son extraordinariamente bajos.

A.5 Implementación del BIDA*

A continuación se presenta una implementación real del BIDA* con profundidad de perímetro 2. Este programa es, en conocimiento del autor de esta Tesis, el más rápido que existe para la resolución de los 100 casos de Richard E. Korf.

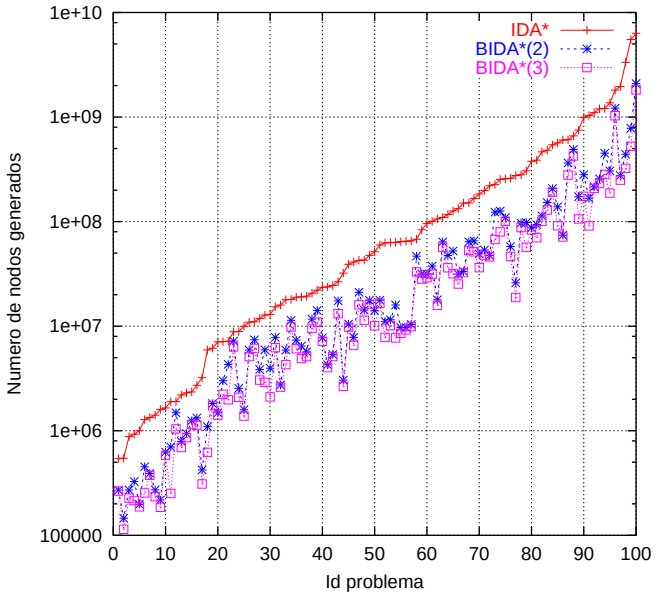


Figura A.8: Número de nodos generados por implementaciones reales del IDA* y el BIDA* en el 15-"Puzle"

Se advierte que el código está desarrollado en C para el dominio del 15-"Puzle". Además, **tiene comentarios de su autor, línea por línea**.

```
/* This program performs bidirectional iterative-deepening A* using a
   perimeter approach on the sliding tile puzzles, using the Manhattan
   distance evaluation function.

   It was written by Carlos Linares, Departamento de Inteligencia Artificial
   Facultad de Informatica, Universidad Politecnica de Madrid, Spain.

   It reuses as much code (and ideas) as possible from ida.c, written by
   Richard E.  Korf, Computer Science Department, University of
   California, Los Angeles, Ca.  90024.                                  */
```

```
#include <stdio.h>                                    /* standard I/O library */
#include <string.h>                                              /* memcpy */
#include <time.h>                                      /* time handling */

#define NUMBER 100              /* number of problem instances to be solved */
#define X 4                                  /* squares in the x dimension */
#define SIZE 16                                 /* total number of squares */
#define MAXPN 4                        /* Maximum number of perimeter nodes */
#define MAXDEPTH 100       /* No prob takes more than 100 plies to be solved */

time_t begin;                                    /* When each search begins */
time_t end;                                        /* When each search ends */

int s[SIZE];                         /* state of puzzle: tile in each position */
int t[SIZE]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};           /* goal state */

int pd=2;                                                  /* Perimeter depth */
int nextPN = 0;                           /* Index to the next perimeter node */
int delta[MAXPN][SIZE];             /* Delta changes for each perimeter node */

int horig[MAXPN]; /* original heuristic estimations until each perimeter node*/
int hests[MAXDEPTH][MAXPN];            /* new heuristic estimations by depth */

struct operators
{int num;                                  /* number of applicable oprs: 2..4 */
 int pos[4];} oprs[SIZE];    /* position of adjacent tiles for each position */

int increment [SIZE] [SIZE] [SIZE];    /* incr eval func: tile, source, dest */

int thresh;                                      /* search cutoff threshold */
double generated;                /* number of states generated per iteration */
double total;                             /* total number of states generated */

/* INITOPS initializes the operator table. */

initops ()

{int blank;                                    /* possible positions of blank */
```

```
 for (blank = 0; blank < SIZE; blank++)  /* for each possible blank position */
   {oprs[blank].num = 0;                                /* no moves initially */
    if (blank > X - 1)                                        /* not top edge */
      oprs[blank].pos[oprs[blank].num++] = blank - X;       /* add a move up */
    if (blank % X > 0)                                        /* not left edge */
      oprs[blank].pos[oprs[blank].num++] = blank - 1;     /* add a move left */
    if (blank % X < X - 1)                                    /* not right edge */
      oprs[blank].pos[oprs[blank].num++] = blank + 1;    /* add a move right */
    if (blank < SIZE - X)                                     /* not bottom edge */
      oprs[blank].pos[oprs[blank].num++] = blank + X;}}   /* add a move down */

/* INIT pre-computes the incremental evaluation function table. For a
given tile moving from a given source position to a given destination
position, it returns the net change in the value of the Manhattan
distance, which is either +1 or -1.  */

init (increment)

int increment [SIZE] [SIZE] [SIZE];/* incr eval function: tile, source, dest */

{int tile;                                                  /* tile to be moved */
 int source, dest;                        /* source and destination positions */
 int destindex;                          /* destination index in operator table */

 for (tile = 0; tile < SIZE; tile++)                      /* all physical tiles */
   for (source = 0; source < SIZE; source++)   /* all legal source positions */
     for (destindex = 0; destindex < oprs[source].num; destindex++)
                                             /* legal destinations of source */
       {dest = oprs[source].pos[destindex];    /* dest is new blank position */
        increment[tile][source][dest]  = abs((tile % X) - (dest % X))
                                       - abs((tile % X) - (source % X))
                                       + abs((tile / X) - (dest / X))
                                       - abs((tile / X) - (source / X));}}

/* INPUT accepts an initial state from the terminal, assuming it is
preceded by a problem number. It stores it in the state vector and
returns the position of the blank tile. */

input (s)
```

```
int s[SIZE];                                                    /* state vector */

{int index;                                          /* index to tile positions */
 int blank;                                           /* position of blank tile */

 scanf ("%*d");                                     /* skip over problem number */
 for (index = 0; index < SIZE; index++)                    /* for each position */
   {scanf ("%d", &s[index]);                      /* input tile in that position */
    if (s[index] == 0) blank = index;}      /* note blank position in passing */
 return (blank);}

/* UPDATEHP returns the new heuristic estimation when moving the tile
   'from' position 'to' position. */

int updatehp (h,hchild,from,to)

     int h[]; /* original heuristic estimations to the active perimeter nodes*/
     int hchild[];                                    /* new heuristic estimates */
     int from;                                  /* Where the tile is pushed from */
     int to;                                         /* Where the tile is placed */

{
  register int i;                                          /* ith perimeter node */
  int value = 1000;              /* Upper bound on the new heuristic distance */

  for (i=0;i<nextPN;i++)                    /* for every active perimeter node */
    if (h[i]<=thresh){         /* only in case it is an active perimeter node */
      if (value>(hchild[i]=h[i]+increment[s[from]+delta[i][s[from]]][from][to]))
        value=hchild[i];}            /* take the minimum heuristic estimation */
  else
    hchild[i]=h[i];
  return value;}                          /* return the new heuristic distance */

/* MANHATTAN returns the sum of the Manhattan distances of each tile,
   except the blank, for reaching the perimeter node 'pidx', plus the
   perimeter depth (pd). */

manhattan (pidx)
```

```
     int pidx;                                  /* Index to the perimeter node */

{
  int value;                                              /* accumulated value */
  int pos;                                                    /* tile position */

  for (value=pos=0; pos<SIZE; pos++)                          /* for each tile */
    if (s[pos] != 0)            /* blank isn't counted in Manhattan distance */
      value +=   abs((pos % X) - ((s[pos]+delta[pidx][s[pos]]) % X))    /* X */
               + abs((pos / X) - ((s[pos]+delta[pidx][s[pos]]) / X));   /* Y */
  return(value+pd);}                              /* Add the perimeter depth */

/* HP returns the minimum manhattan distance to all the perimeter
   nodes from s.

   This function is employed only for getting the original heuristic
   distances from the start state to all perimeter nodes. Thus, it
   makes use of the global variable s and assumes all perimeter nodes
   to be active. */

int hp ()

{
  register int i;                                       /* ith perimeter node */
  int value = 1000;         /* upper bound on the minimum heuristic distance */

  for (i=0;i<nextPN;i++)            /* for all the generated perimeter nodes */
    if (value >= (horig[i]=manhattan (i))) /* compute its heuristic distance */
      value=horig[i];                                 /* and take the minimum */
  return value;}                    /* return the minimum heuristic distance */

/* GENERATEPERIMETER computes the delta changes for all the perimeter
   nodes at depth pd. It follows the same expansion procedure employed
   by Korf in his IDA* implementation */

void generateperimeter (blank, oldblank, pd)

     int blank;                                  /* current position of blank */
```

```
     int oldblank;                              /* previous position of blank */
     int pd;                                              /* perimeter depth */

{
  int newblank;                               /* blank position in new state */
  register int i;                               /* ith position of the board */

  if (!pd) {                                    /* is this a perimeter node? */
    t[blank]=0;                              /* place the blank in its position */
    for (i=0 ; i<SIZE ; delta[nextPN][t[i]] = i-t[i++]);/* Compute its delta */
    nextPN++;}                /* update the index to the next perimeter node */
  else {          /* it is not a perimeter node, so, generate its successors */
    for (i=0 ; i<oprs[blank].num; i++) {     /* for each applicable operator */
      if ((newblank = oprs[blank].pos[i]) != oldblank) {  /* not inv lst mov */
        t[blank]=t[newblank];                              /* make actual move */
        generateperimeter (newblank,blank,pd-1);                /* go deeper */
        t[newblank]=t[blank];}}}}     /* undo current move before doing next */

/* SEARCH performs one depth-first iteration of the search using a
 perimeter whose heuristic distances from s are written in h, cutting
 off when the depth plus the heuristic evaluation exceeds THRESH. If
 it succeeds, it returns 1 and records the sequence of tiles moved in
 the solution.  Otherwise, it returns 0 */

search (h,blank,oldblank,g)

int h[];      /* current heuristic estimation for each active perimeter node */
int blank;                                       /* current position of blank */
int oldblank;                                  /* previous position of blank */
int g;                                             /* current depth of search */

{ int index;                                     /* index into operator array */
  int newblank;                               /* blank position in new state */
  int newh;                                 /* heuristic evaluation of new state */

  for (index = 0; index < oprs[blank].num; index++)     /* for each appl opr */
    if ((newblank = oprs[blank].pos[index]) != oldblank){/*not inv last move */
      newh=updatehp(h,hests[g],newblank,blank); /* compute new heuristic est */
      generated++;                                    /* count nodes generated */
```

```
      if (newh+g+1 <= thresh) {                    /* less than search cutoff */
        s[blank]=s[newblank];                            /* make actual move */
        if ((newh == pd) ||                             /* collision detected! */
            (search (hests[g],newblank,blank,g+1)))       /* search succeeds */
          return (1);                                    /* exit with success */
        s[newblank]=s[blank];}}       /* undo current move before doing next */
  return (0);}                                           /* exit with failure */

/* Main program does the initialization, inputs an initial state,
   solves it, and prints the solution, along with some statistics:
   time spent, total time and the mean time */

main ()

{int success;                           /* boolean flag for success of search */
 int blank;                                      /* initial position of blank */
 int initeval;                          /* manhattan distance of initial state */
 int problem;                                        /* problem instance */
 int index;                                     /* index to tile positions */
 long totalTime = 0;        /* time elapsed for solving the whole test suite */

 initops ();                                     /* initialize operator table */
 init (increment);                          /* initialize evaluation function */

 for (problem = 1; problem <= NUMBER; problem++) { /* for each initial state */
   begin = time ((time_t*) 0);    /* ready? anyway, I am gonna call you!! :) */
   blank = input(s);                                 /* input initial state */
   nextPN = 0;                             /* index to the next perimeter node */
   generateperimeter (0,-1,pd);        /* generate the perimeter at depth pd */
   thresh = initeval = hp();              /* initial threshold is initial hp */
   total = 0;                              /* initialize total nodes generated */

   do {                    /* depth-first iterations until solution is found */
     generated = 0;              /* initialize number generated per iteration */
     success = search (horig,blank,-1,0);                   /* perform BIDA* */
     printf ("%d %10.f\n", thresh, generated);        /* nodes per iteration */
     fflush(stdout);        /* flush output buffer to see progress of search */
     total = total + generated;     /* keep track of total nodes per problem */
     thresh += 2;}                      /* threshold always increases by two */
```

```
while (!success);                                /* until solution is found */

end = time ((time_t*)0);                                  /* Stop the clock! */

totalTime += (end-begin);                  /* compute the total time spent */
printf ("%d %d %10.f %d/%d (%g)\n",
         problem, thresh-2, total, end - begin,
         totalTime,totalTime/(problem*1.0));}}
```

A.5.1 Comentarios al código del BIDA*

El código mostrado en la sección precedente utiliza las mismas tablas precomputadas que se usan en la implementación del IDA* de Richard E. Korf:

- Tabla `oprs` para la expansión rápida de los sucesores de un nodo.
- Tabla `increment` para la evaluación heurística rápida de cada nodo.

Mientras que la tabla `oprs` se emplea del mismo modo en el código del BIDA*, que en el IDA*, la segunda tabla requiere alguna computación adicional para su utilización, puesto que en el caso del IDA*, dicha tabla se precomputaba sabiendo de antemano que el nodo destino sería siempre el estado mostrado en la figura A.9(a), mientras que en el caso del BIDA*, las distancias heurísticas se miden hasta cualquiera de los nodos de perímetro, m_j.

Dicha computación adicional consiste en la recolección de alguna información durante la generación de los nodos de perímetro, para saber cómo se transforma el estado final, `t`, en cada uno de los nodos de perímetro generados.

Denominando $\rho(i)$ al contenido de la posición i en un tablero cualquiera, n, la posición deseada de la ficha que hay en la posición i es siempre $\rho(i)$, para el estado final de la figura A.9(a). Por lo tanto, para medir la diferencia heurística desde un nodo cualquiera, n, hasta un nodo de perímetro, m_j, basta con enmascarar este valor para que devuelva la posición deseada de la ficha que hay en la posición i, en el nodo m_j. Definiendo:

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Nodo final, t

	1	9	7
11	13	5	3
14	12	4	2
8	6	10	15

(b) Nodo n

1	5	2	3
8	4	6	7
	9	10	11
12	13	14	15

(c) Nodo m_j

8	-1	0	0
1	-4	0	0
-4	0	0	0
0	0	0	0

(d) $\delta_{m_j}(\cdot)$

Figura A.9: Cálculo optimizado de la distancia heurística hasta cualquier nodo

$$\rho'(i) = \rho(i) + \delta_{m_j}(\rho(i)) \tag{A.7}$$

donde $\delta_{m_j}(\rho(i))$ es la diferencia entre la posición actual de la ficha $\rho(i)$ en el nodo de perímetro m_j (es decir, $\rho'(i)$), y su posición deseada en el estado final, `t`, que será, precisamente, $\rho(i)$. Es decir:

$$\delta_{m_j}(\rho(i)) = \rho'(i) - \rho(i) \tag{A.8}$$

que resulta simplemente de despejar el valor $\delta_{m_j}(\rho(i))$, de la ecuación (A.7). La conveniencia de este cálculo es que el vector de valores $\delta_{m_j}(\cdot)$ es muy fácil de calcular, según la expresión (A.8), cada vez que se ha generado un nuevo nodo de perímetro, tal y como se hace en la función `generateperimeter`.

Así, la ecuación (3.1), mostrada en la página 83 de la Tesis, para el cálculo de la distancia heurística hasta un nodo final tan convenientemente elegido, se modifica ahora ligeramente para incluir este término adicional de la siguiente manera:

$$\begin{aligned} h(n) = & \sum_{\rho(i)=1}^{N^2-1} \left(\left| \frac{\rho'(i)}{N} - \frac{i}{N} \right| + |((\rho'(i)) \bmod N) - (i \bmod N)| \right) \\ = & \sum_{\rho(i)=1}^{N^2-1} \left| \frac{\rho(i)+\delta_{m_j}(\rho(i))}{N} - \frac{i}{N} \right| + \\ & + \left| ((\rho(i) + \delta_{m_j}(\rho(i))) \bmod N) - (i \bmod N) \right| \end{aligned} \tag{A.9}$$

La figura A.9 muestra un ejemplo. En este caso, el algoritmo de búsqueda aplicado en la búsqueda hacia delante, necesita calcular la distancia heurística desde el nodo n, mostrado en la figura A.9(b), hasta el nodo de perímetro, m_j, que se indica en la figura A.9(c). Durante la generación del perímetro, la función `generateperimeter` calculó el vector $\delta_{m_j}(\cdot)$, según la expresión (A.8), mostrado en la figura A.9(d), de modo que la distancia heurística es, según la expresión (A.9) igual a:

$$\begin{aligned} h(n, m_j) = & (\quad\quad) + (\mathbf{0} + \mathbf{1}) + (2 + 1) + (1 + 0) \\ & (1 + 3) + (2 + 0) + (\mathbf{1} + \mathbf{1}) + (1 + 0) \\ & (1 + 2) + (1 + 1) + (\mathbf{1} + \mathbf{1}) + (2 + 1) \\ & (\mathbf{2} + \mathbf{0}) + (2 + 1) + (1 + 0) + (0 + 0) = 30 \end{aligned}$$

Nótese que la primera posición, está intencionadamente vacía, puesto que en el cálculo de la distancia de Manhattan, se ignora siempre el blanco, y es en la posición marcada al efecto, en la expresión anterior, donde se encuentra el blanco en el nodo n, según la figura A.9(b). Por otra parte, las posiciones marcadas en negrita son las únicas que resultan afectadas por los valores de $\delta_{m_j}(\cdot)$.

Gracias al uso del vector $\delta_{m_j}(\cdot)$, es posible utilizar la misma tabla precomputada `increment`, que empleó Richard E. Korf en el código del IDA*, de modo que:

`increment` `[`$\rho(i) + \delta_{m_j}(\rho(i))$`]` `[`$i$`]` `[`$k$`]`

devuelve el incremento en la evaluación heurística al mover la ficha que hay en la posición i, en el tablero n, cuyo contenido es $\rho(i)$, hasta la posición k —cuyo contenido debe ser necesariamente, $\rho(k) = 0$, puesto que si no estuviera el blanco en esa posición, el movimiento no sería legal. Ésta es, exactamente, la forma en que se emplea la tabla `increment` en la función `updatehp`, que es la encargada de calcular el nuevo valor heurístico de un sucesor del nodo actualmente expandido, según la expresión (A.1).

Puesto que el método descrito sirve para calcular las nuevas estimaciones heurísticas de cada nodo hasta los nodos activos del perímetro, como la diferencia con el valor de su padre, es preciso calcular las estimaciones originales. Este cálculo se hace una sola

vez, para cada problema a resolver, en la función `hp`, que utiliza la función `manhattan` modificada, según la ecuación (A.9).

Además, existen otras optimizaciones propias de una implementación real de perímetro:

- La detección de una colisión se hace, simplemente, comprobando que el nuevo valor de la función heurística es, exactamente, igual a la profundidad de perímetro, p_d, como se hace en la función `search`.
- El estado (activo o inactivo) de cada nodo de perímetro no tiene por qué almacenarse en un vector dedicado a ello, puesto que todos los nodos de perímetro, `i`, para los que su distancia heurística desde el nodo original, `h[i]`, sea menor o igual que el umbral actual, `thresh`, están activos; el resto están inactivos.

  Esta es, precisamente, la comprobación que se hace en la función `updatehp`.
- Una optimización significativa consiste en evitar crear vectores para almacenar las nuevas distancias heurísticas de cada sucesor del nodo actualmente explorado, hasta los nodos activos de perímetro. Puesto que los nodos se exploran en un orden de el primero en profundidad, es posible crear una matriz, como aquella llamada `hests`, que contiene, por cada nivel del árbol de búsqueda, las estimaciones heurísticas hasta cada uno de los nodos de perímetro activos.

  Su uso sirve para evitar que `search` tenga que crear variables temporales cuya construcción consume mucho tiempo (más aún si se considera que el número de llamadas recursivas a esta función puede ser de varios cientos o miles de millones de veces), haciendo copias en direcciones de memoria previamente creadas y cuyos contenidos pueden desecharse cuando la llamada recursiva retorna.

Por último, se advierte que para probar nuevos valores de perímetro, es preciso:

- Modificar el valor de la variable global `pd`, a la profundidad de perímetro deseada.
- Modificar el valor de la definición `MAXPN` para que sea igual al número de nodos generados a la profundidad de perímetro escogida. Para $p_d = 3$, `MAXPN` debe valer 10, y para $p_d = 4$, debe ser igual a 24. En otros casos, si se ignora el

número de nodos de perímetro generados a una profundidad arbitraria, puede inicializarse a un valor muy alto, pero se advierte que ello consume un tiempo innecesario en la creación de posiciones de memoria que serán inútiles para los vectores `delta[MAXPN][SIZE]`, `horig[MAXPN]` y `hests[MAXDEPTH][MAXPN]`.

A.6 Resultados de las pruebas en el N-"Puzle"

A continuación se muestran los resultados de los mismos algoritmos probados en la Tesis Doctoral sobre las primeras 50 instancias del juego de ensayo de Richard E. Korf —página 155 de la Tesis Doctoral.

A.6.1 Leyenda

En las tablas que se muestran a continuación se han usado los siguientes símbolos en las cabeceras:

No.	Número de instancia.
#h	Número de evaluaciones heurísticas.
#F	Número de actualizaciones de F.
#p	Número de nodos de perímetro generados.
t	Tiempo de ejecución (en segundos).
#g	Número de nodos generados.
#e	Número de nodos expandidos.
b	Factor de ramificación.

El uso de estas abreviaciones mejorará sensiblemente la presentación de las tablas.

A.6.2 IDA*

Tabla A.1: Resultados del IDA* con el 15-"Puzle"

No.	#h	t	#g	#e	b
00	798261	47	798292	388965	2.05234
01	249393	16	249423	123119	2.02585
02	1022778	63	1022805	513650	1.99125
03	396987	24	397023	196310	2.02242
04	2483432	151	2483456	1224209	2.02862
05	2845728	171	2845758	1396700	2.03749
06	253060	15	253092	129590	1.95301
07	3337442	205	3337479	1645232	2.02858
08	7821457	473	7821490	3858456	2.0271
09	1470165	90	1470198	719230	2.04412
10	2392898	149	2392936	1216935	1.96636
11	1568885	98	1568912	776138	2.02143
12	1905098	113	1905128	952869	1.99936
13	2905603	177	2905631	1422937	2.04199
14	2274134	136	2274164	1164498	1.95291
15	1158086	68	1158112	567494	2.04074
16	1355560	82	1355589	681316	1.98966
17	10384429	641	10384462	5391316	1.92615
18	9519007	586	9519041	4821834	1.97415
19	4974159	310	4974189	2448039	2.03191
20	17634583	1081	17634615	8897062	1.98743
21	9630805	587	9630838	4913723	1.95999
22	8415318	518	8415347	4219383	1.99445
23	4776954	295	4776980	2356637	2.02703
24	19478869	1241	19478906	9972768	1.97407
25	6515769	403	6515803	3322180	1.9613
26	17537736	1089	17537773	9052255	1.94127
27	18080016	1122	18080040	9075720	1.99721

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
28	13991933	830	13991958	6880956	2.03343
29	6474701	403	6474736	3198347	2.0244
30	11770340	741	11770371	6066422	1.94025
31	19781224	1214	19781254	9856181	2.028
32	31820183	1995	31820217	15796728	2.05212
33	5839191	348	5839215	2819572	2.07096
34	37577326	2371	37577359	19312448	1.90063
35	14104613	858	14104638	6968764	2.02398
36	12821472	841	12821502	6372867	2.01189
37	24819593	1629	24819618	12434788	2.02672
38	12694924	780	12694959	6253507	2.03005
39	76972815	5346	76972857	38747644	1.65976
40	40386517	2571	40386549	20783313	1.87137
41	19610682	1217	19610715	10150994	1.95208
42	63161363	4030	63161396	31939192	1.73259
43	34861100	2175	34861132	17216351	2.05185
44	71891984	4442	71892021	35585400	1.74274
45	63563776	4089	63563810	33933472	1.50496
46	41729677	2934	41729712	20919629	1.90055
47	49729320	3367	49729354	24122935	1.90752
48	85204432	5584	85204468	43450457	1.54449
49	83548898	5529	83548938	41102998	1.6327

A.6.3 RBFS

Tabla A.2: Resultados del RBFS con el 15-"Puzle"

No.	#h	#F	t	#g	#e	b
00	798191	388875	68	798190	388917	2.05234
01	456973	225528	38	456972	225572	2.02583

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
02	292814	147248	25	292813	147288	1.98802
03	711714	351711	59	711713	351752	2.02333
04	2978175	1469380	261	2978174	1469424	2.02676
05	1430964	701787	130	1430963	701831	2.0389
06	498242	254020	45	498241	254073	1.96101
07	3742048	1848326	324	3742047	1848373	2.02451
08	2568646	1273201	223	2568645	1273247	2.0174
09	2912455	1426323	256	2912454	1426368	2.04187
10	6525212	3318570	589	6525211	3318625	1.96624
11	1568847	776054	137	1568846	776103	2.02144
12	1905039	952775	163	1905038	952822	1.99936
13	1351973	660592	119	1351972	660633	2.04648
14	2274053	1164398	208	2274052	1164448	1.9529
15	2139878	1055833	183	2139877	1055877	2.02663
16	2753979	1373732	251	2753978	1373781	2.00467
17	11204238	5813692	994	11204237	5813743	1.9272
18	9518915	4821725	849	9518914	4821776	1.97415
19	3336406	1642480	300	3336405	1642520	2.03127
20	17634480	8896944	1615	17634479	8896994	1.98731
21	15010075	7635036	1328	15010074	7635086	1.96593
22	10430987	5237604	927	10430986	5237654	1.99154
23	6021388	2971922	521	6021387	2971965	2.02606
24	8569982	4355146	773	8569981	4355195	1.96776
25	10890366	5555558	1056	10890365	5555610	1.96025
26	13342545	6855238	1269	13342544	6855289	1.94631
27	12385612	6209529	1177	12385611	6209575	1.9946
28	23694457	11611674	2118	23694456	11611721	2.06755
29	7809213	3863221	696	7809212	3863272	2.0214
30	19877314	10239193	1926	19877313	10239247	1.95033
31	11579435	5797963	1019	11579434	5798010	1.99714

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
32	20602407	10216461	1863	20602406	10216511	2.03056
33	15755630	7606826	1385	15755629	7606867	2.07124
34	18398328	9443326	1645	18398327	9443376	1.95438
35	8746051	4321662	754	8746050	4321707	2.02375
36	13051279	6486889	1141	13051278	6486934	2.01193
37	20271090	10135676	1781	20271089	10135726	2.01685
38	9698776	4794596	847	9698775	4794640	2.02284
39	76972749	38747533	7248	76972748	38747590	1.65535
40	10582144	5423030	965	10582143	5423082	1.95131
41	19610610	10150894	1756	19610609	10150947	1.95186
42	25481864	12954505	2327	25481863	12954556	2.02361
43	34860972	17216220	3279	34860971	17216272	2.04978
44	35559514	17541821	3162	35559513	17541870	2.03141
45	63563642	33933319	5891	63563641	33933383	1.50736
46	43281592	21665089	4150	43281591	21665136	1.88524
47	45736001	22205483	4008	45736000	22205528	1.93087
48	43830423	22530119	3942	43830422	22530172	1.81821
49	51730824	25549661	4833	51730823	25549712	1.82947

A.6.4 BIDA*

El tamaño del perímetro a profundidad 4 resultó ser 24 nodos.

Tabla A.3: Resultados del $BIDA^*_4$ con el 15-"Puzle"

No.	#h	t	#g	#e	b
00	715846	23	205442	97069	2.11651
01	111059	3	32562	15570	2.09172
02	962214	29	243152	119325	2.0378
03	319505	10	90632	43944	2.06261
04	1390012	45	383195	184685	2.07489

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
05	1767059	56	485416	233324	2.08047
06	770744	20	131569	65821	1.99904
07	1586952	50	424745	205082	2.07113
08	3641108	119	924488	450457	2.05235
09	1666360	53	462429	221725	2.08563
10	1169955	34	248879	127274	1.95555
11	1938123	62	532586	256553	2.07596
12	1255162	42	366147	181388	2.01864
13	4192641	133	1143743	543080	2.10604
14	3302664	108	940956	466168	2.01851
15	1375188	44	387053	184217	2.1011
16	377937	12	104879	52168	2.01059
17	2406932	75	613709	318859	1.92474
18	6309343	242	2248119	1110804	2.02387
19	4277102	115	880251	421642	2.08769
20	23106421	670	5054140	2492191	2.02799
21	9173459	274	2215748	1097701	2.01854
22	19443837	607	4953752	2439091	2.03099
23	4698197	132	993081	489277	2.02971
24	7870999	272	2228202	1121903	1.9861
25	5554337	209	1784969	887728	2.01073
26	25561257	818	6934921	3477102	1.99446
27	11300231	394	3477463	1699025	2.04675
28	7868993	287	2635764	1289244	2.04443
29	4630120	121	851445	418492	2.03458
30	11844225	475	4185223	2121543	1.97273
31	17934201	459	3474901	1684589	2.06276
32	16255934	562	5127683	2484059	2.06424
33	10294841	319	2803826	1307447	2.14451
34	49188178	1543	12943539	6502591	1.99052

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
35	9119383	292	2581327	1245771	2.07208
36	13401813	371	2941014	1420369	2.0706
37	40033138	1192	9910765	4867296	2.0362
38	9305005	381	3848288	1840574	2.09081
39	45182563	1442	11431807	5607665	2.03861
40	16115111	543	4791013	2388902	2.00553
41	10978751	391	3605784	1838540	1.96123
42	77098777	3072	27184310	13390403	2.08589
43	7135168	254	1986240	967162	2.05369
44	36546804	1200	9754197	4758002	2.05006
45	25082208	842	6349694	3334971	1.90398
46	46426102	1558	12610388	6216202	2.02863
47	55829898	1595	11971815	5638867	2.12309
48	50061828	1967	18258589	9162893	2.00056
49	43708900	1246	9731606	4672050	2.08294

El tamaño del perímetro a profundidad 6 resultó ser de 107 nodos.

Tabla A.4: Resultados del $BIDA^*_6$ con el 15-"Puzle"

No.	#h	t	#g	#e	b
00	613916	20	130292	61522	2.11762
01	157940	4	29042	13954	2.08094
02	997974	28	164415	80211	2.0498
03	305993	9	54647	26451	2.06592
04	1926971	56	324700	157860	2.05689
05	1703499	51	308339	147745	2.08693
06	546188	17	100558	50230	2.00218
07	2728190	72	386525	187355	2.06306
08	2850637	89	571417	279488	2.04452

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
09	1340266	45	298834	143909	2.07652
10	1550471	41	223469	114229	1.95651
11	1975252	61	383313	184756	2.07468
12	1283307	35	206878	100818	2.05201
13	3653549	131	880070	422293	2.08401
14	2792606	86	547036	269197	2.03212
15	1490853	44	251472	118392	2.12395
16	612048	16	98604	49188	2.00486
17	2798525	83	483297	250021	1.93313
18	5954098	180	1104072	548748	2.012
19	4344026	118	681295	327282	2.08166
20	19590775	552	3172871	1549298	2.04794
21	10957401	306	1748191	862334	2.02729
22	18508655	530	2909841	1427281	2.03873
23	4263304	119	658250	321822	2.04539
24	9004988	282	1736560	876124	1.98211
25	5441212	192	1252510	617366	2.02881
26	27139077	816	4868785	2422857	2.00953
27	11064367	345	2150638	1059481	2.0299
28	6572233	185	1070629	524984	2.03936
29	4643230	111	535188	264233	2.02547
30	6486471	223	1382784	697194	1.98338
31	17988652	509	2968814	1446684	2.05215
32	23066018	723	4129181	2016479	2.04772
33	12383194	340	1975324	923480	2.13898
34	51600774	1531	8974372	4494117	1.99692
35	7907029	248	1587193	765040	2.07465
36	14044517	357	1890279	913470	2.06934
37	34391889	949	5409778	2652965	2.03915
38	9653086	351	2427014	1163434	2.08607

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
39	47915950	1438	8380557	4097604	2.04523
40	22995839	655	3567363	1783712	1.99997
41	14079266	471	3046729	1536556	1.98284
42	128149726	4153	25179334	12401024	2.07799
43	9935530	271	1527473	745819	2.04805
44	44493338	1262	7290226	3538965	2.05999
45	30380148	927	5419763	2832033	1.91375
46	44929994	1286	7495452	3695362	2.02834
47	56496928	1554	8945901	4236784	2.11148
48	41803758	1290	7723426	3842658	2.00992
49	46838520	1189	6144500	2952090	2.08141

El tamaño del perímetro a profundidad 8 resultó ser de 454 nodos.

Tabla A.5: Resultados del $BIDA^*_8$ con el 15-"Puzle"

No.	#h	t	#g	#e	b
00	668681	29	90463	42605	2.12384
01	177666	7	23672	11463	2.07121
02	1083302	40	125366	61017	2.05598
03	250331	11	37045	17538	2.11409
04	1437833	52	159951	77442	2.06639
05	1848098	72	233839	111684	2.0942
06	475178	18	58380	28730	2.03558
07	2115568	70	207048	99780	2.07571
08	2551843	103	343397	168313	2.04081
09	1181457	46	154263	74201	2.07983
10	1251534	44	135842	68822	1.97607
11	1926011	79	260596	125368	2.07915
12	1353746	48	149448	72858	2.05242

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
13	3407426	161	573309	273602	2.09559
14	2797650	108	353482	174793	2.02294
15	1520197	55	171666	80604	2.12997
16	463029	22	74665	37233	2.00873
17	3152281	113	350456	179969	1.94831
18	6641015	271	881487	435869	2.02263
19	4583332	158	483782	232705	2.07922
20	19875227	700	2117926	1031069	2.05419
21	10994799	402	1260119	620393	2.03133
22	19124515	649	1902589	926168	2.05435
23	3886423	139	432750	209917	2.0619
24	7844253	312	1030372	514713	2.00209
25	5782504	250	845348	414031	2.04198
26	25115912	1039	3423232	1704578	2.00833
27	9739355	420	1412692	685313	2.0615
28	6298778	225	691739	337499	2.04987
29	3271488	117	350033	171343	2.04343
30	5367510	244	838988	421336	1.99159
31	14349097	507	1567226	758298	2.06687
32	20342451	784	2533931	1228713	2.06233
33	10421526	357	1091360	510662	2.13717
34	55427652	2144	6799754	3395399	2.00268
35	8488276	351	1159655	558243	2.07745
36	10701167	360	1034484	500917	2.06533
37	30622910	1065	3229620	1583302	2.03986
38	8830652	384	1319172	637640	2.06895
39	53731033	1919	5806013	2832774	2.04962
40	18194540	789	2693503	1337009	2.01466
41	13162105	603	2126720	1062088	2.00252
42	93643416	4419	15453142	7561566	2.04366

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
43	10555082	435	1227084	597335	2.05441
44	52448639	1788	5435520	2623874	2.07159
45	26786446	1194	3401258	1776335	1.91488
46	44104700	1611	4617039	2255105	2.04741
47	48300898	1664	4676159	2203442	2.12222
48	37965101	1447	4574532	2268172	2.01689
49	42537933	1526	4445305	2123024	2.09388

A.6.5 RBFPS*

El tamaño del perímetro a profundidad 4 resultó ser 24 nodos.

Tabla A.6: Resultados del RBFPS$_4^*$ con el 15-"Puzle"

No.	#h	#F	t	#g	#e	b
00	342776	47215	14	100168	47251	2.12002
01	120012	16695	4	34982	16736	2.09059
02	962064	119259	35	243085	119297	2.03772
03	431774	59616	17	123327	59653	2.06753
04	1389893	184616	54	383134	184656	2.07489
05	1966576	262709	77	546729	262751	2.08081
06	778351	66386	23	132792	66435	1.99898
07	1586970	205019	59	424700	205062	2.07112
08	3640747	450386	132	924424	450428	2.05234
09	1684678	219022	71	454362	219063	2.07415
10	1526175	163668	50	318917	163720	1.94802
11	1937863	256469	74	532511	256514	2.07598
12	1707229	259300	70	521303	259343	2.01013
13	2005572	253164	75	535748	253201	2.11592
14	4893673	688708	192	1384955	688754	2.01083

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
15	2446431	336740	95	703085	336780	2.08769
16	377704	52098	15	104828	52143	2.01058
17	2329846	320390	88	616926	320437	1.92531
18	5362634	941424	251	1903336	941471	2.02167
19	4890622	488736	155	1019594	488777	2.08602
20	8434547	935483	291	1899415	935527	2.03032
21	5152403	588239	177	1187680	588280	2.01892
22	22999348	2937892	897	5966520	2937938	2.03086
23	5287002	553536	174	1123312	553576	2.02921
24	3670919	565544	157	1126729	565589	1.99215
25	7456955	1214284	333	2432658	1214333	2.0033
26	21840289	2970027	862	5954942	2970074	2.00498
27	7560950	1182838	326	2425885	1182880	2.05084
28	7014655	1185129	330	2415360	1185172	2.03799
29	7232263	639362	215	1296936	639409	2.02835
30	18942224	3480627	913	6861605	3480677	1.97135
31	10111478	989937	325	2027819	989980	2.04835
32	15596838	2557983	703	5271815	2558029	2.06089
33	26378857	3293707	990	7055549	3293744	2.14211
34	24144675	3209033	920	6360282	3209081	1.98197
35	8684380	1252308	356	2595745	1252349	2.07271
36	15783520	1707093	541	3527036	1707137	2.06606
37	26783988	3324516	982	6789189	3324562	2.04213
38	7073439	1506190	383	3137556	1506230	2.08306
39	20805989	2599158	800	5276939	2599209	2.03021
40	16114572	2388818	700	4790946	2388869	2.00553
41	7879682	1381744	365	2724433	1381792	1.97167
42	31652648	5864665	1561	11865328	5864712	2.02317
43	7134410	967061	277	1986131	967109	2.05369
44	18482953	2326133	734	4775987	2326178	2.05315

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
45	15053476	2113819	591	4039736	2113870	1.91107
46	69655892	9504920	2795	19241661	9504969	2.03609
47	54660023	5488159	1793	11646642	5488199	2.12213
48	26297298	4897093	1319	9745861	4897142	1.99011
49	39439507	4029674	1306	8399018	4029722	2.08427

El tamaño del perímetro a profundidad 6 resultó ser de 107 nodos.

Tabla A.7: Resultados del $RBFPS^*_6$ con el 15-"Puzle"

No.	#h	#F	t	#g	#e	b
00	297741	30213	11	64244	30247	2.12354
01	172046	15043	6	31385	15082	2.08066
02	997725	80147	32	164348	80183	2.04968
03	412815	35820	14	74175	35855	2.06869
04	1926886	157793	62	324639	157831	2.05688
05	1123093	106169	42	222245	106209	2.09246
06	552944	50701	20	101577	50748	2.00183
07	2728829	187294	80	386480	187335	2.06304
08	2850446	279420	103	571354	279460	2.0445
09	1366975	144655	52	298337	144694	2.06184
10	2012543	144826	59	282593	144876	1.95074
11	1975713	184674	70	383238	184717	2.07471
12	1722426	138292	55	282712	138333	2.04373
13	1775912	204588	73	428417	204623	2.09365
14	5566171	528059	196	1069993	528103	2.02612
15	2716792	215336	91	455401	215374	2.11441
16	612125	49120	19	98553	49163	2.00484
17	2604650	240123	89	463678	240168	1.93075
18	5579115	501141	194	1010523	501186	2.01628

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
19	4928284	377517	153	785313	377556	2.07998
20	6955507	558325	228	1141833	558367	2.04496
21	6270324	463485	191	939190	463524	2.02621
22	21984318	1708284	698	3485052	1708328	2.04004
23	4756503	360724	146	737296	360762	2.04373
24	3947965	425585	161	844450	425628	1.98405
25	7124356	816809	290	1655379	816856	2.02653
26	27137096	2422764	940	4868705	2422814	2.00953
27	7815056	758562	284	1545954	758602	2.0379
28	6016249	485594	193	986322	485635	2.03101
29	6268794	358013	167	725419	358058	2.026
30	7466174	798692	287	1585386	798739	1.98488
31	10182422	860342	350	1758658	860383	2.04404
32	21899110	2032162	818	4152645	2032206	2.04342
33	32211993	2326430	995	4965484	2326465	2.13434
34	25029924	2244355	856	4462955	2244401	1.98849
35	7648294	766231	284	1592729	766270	2.07854
36	16519536	1086942	496	2243733	1086984	2.06418
37	23127558	1886440	751	3856521	1886484	2.04429
38	10148114	1234723	433	2572730	1234766	2.08357
39	22385399	1928544	798	3929836	1928593	2.03767
40	22994658	1783630	752	3567296	1783679	1.99997
41	10477042	1186138	418	2364847	1186184	1.99367
42	57088427	5528664	2091	11176878	5528709	2.02161
43	9933298	745726	307	1527377	745772	2.04805
44	22436845	1710449	702	3528126	1710492	2.06264
45	17884417	1808743	651	3473887	1808792	1.92057
46	68655189	5721714	2285	11591293	5721761	2.02583
47	54025304	4067433	1726	8587734	4067471	2.11132
48	37332529	3394139	1361	6832185	3394185	2.01291

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
49	30770044	1948251	878	4048570	1948296	2.078

El tamaño del perímetro a profundidad 8 resultó ser de 454 nodos.

Tabla A.8: Resultados del RBFPS_8^* con el 15-"Puzle"

No.	#h	#F	t	#g	#e	b
00	323406	21172	14	45156	21204	2.13042
01	199472	12421	8	25729	12458	2.07093
02	1085317	60955	42	125299	60989	2.05583
03	350960	24290	17	51412	24323	2.115
04	1439509	77377	55	159890	77413	2.06638
05	1184299	77779	54	163718	77817	2.10441
06	480952	28976	20	58925	29021	2.034
07	2121375	99723	76	207006	99762	2.07566
08	2554924	168249	111	343339	168287	2.04078
09	1198460	74295	52	153512	74332	2.06623
10	4442305	219276	159	435839	219323	1.98786
11	1928895	125294	86	260532	125335	2.07919
12	1356746	72797	51	149406	72836	2.05246
13	1579538	134368	86	283830	134401	2.11207
14	2802529	174714	117	353397	174756	2.02288
15	2823929	147581	107	312842	147617	2.11947
16	465190	37167	22	74614	37208	2.00871
17	2848579	173498	114	336459	173541	1.93987
18	6325885	411467	272	833119	411510	2.02481
19	5194465	268200	195	556773	268237	2.07592
20	7226329	385884	275	790922	385924	2.04966
21	6058138	329475	233	668740	329512	2.02981
22	22931328	1128138	838	2315056	1128180	2.0521

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
23	4332642	234730	167	483540	234766	2.06001
24	3267908	236630	152	473167	236671	1.99983
25	5785682	413956	272	845287	414001	2.04198
26	25112050	1704487	1194	3423152	1704535	2.00834
27	7027769	501131	329	1035769	501169	2.06685
28	10722569	557629	406	1150697	557668	2.06355
29	4479094	240217	171	491474	240260	2.04598
30	6165810	469524	297	935200	469569	1.99191
31	8297128	500673	343	1032108	500712	2.06144
32	19215994	1229920	858	2533652	1229962	2.06001
33	27350880	1296643	987	2767347	1296676	2.1342
34	26823853	1681414	1168	3355156	1681458	1.99547
35	8146377	557184	409	1157834	557221	2.07799
36	12685464	602986	494	1241902	603026	2.05958
37	19985313	1064006	846	2170259	1064048	2.03972
38	9336496	681072	467	1407701	681113	2.06687
39	24612529	1358831	967	2777109	1358878	2.04375
40	18194050	1336929	917	2693436	1336976	2.01466
41	9593689	839073	513	1691379	839117	2.01581
42	40405717	3234752	2047	6572622	3234795	2.03188
43	10551429	597244	416	1226988	597288	2.05441
44	27099663	1272170	955	2640777	1272211	2.07579
45	15968043	1141927	730	2194650	1141974	1.92198
46	47714851	2501039	1817	5137439	2501078	2.05412
47	45287987	2059413	1579	4372744	2059450	2.12327
48	33644811	1929348	1350	3897805	1929392	2.02028
49	28438143	1452116	1066	3036554	1452159	2.0911

A.6.6 BRBFS*

Tabla A.9: Resultados del BRBFS* con el 15-"Puzle"

No.	#h	#F	#p	t	#g	#e	b
00	1086414	14874	14823	45	31566	14902	2.1181
01	449870	7896	7521	17	16312	7930	2.05674
02	1118423	12452	10922	36	25395	12485	2.03388
03	262006	5551	5560	8	11511	5575	2.06438
04	1297530	14280	13313	56	29123	14314	2.03444
05	1096435	15015	12207	43	30995	15039	2.06084
06	492790	15187	11028	40	30319	15212	1.99297
07	1968493	19466	16679	69	39701	19501	2.03574
08	2751108	41718	34249	182	85431	41743	2.04655
09	2393944	25356	22681	154	52184	25397	2.05465
10	779088	15099	12046	43	29548	15144	1.95101
11	1518897	17694	16153	80	36386	17725	2.05269
12	1308372	13029	12428	45	26593	13057	2.03653
13	1531039	23323	19063	72	48524	23359	2.07723
14	1786745	20449	17913	69	40962	20489	1.99912
15	1666688	15473	15030	70	32221	15505	2.07797
16	486081	8235	7397	17	16832	8267	2.0358
17	3665053	38837	32063	306	77544	38871	1.99485
18	6131986	58754	48145	449	117768	58790	2.00316
19	4439732	29353	27063	212	60508	29386	2.05901
20	16122836	82236	69777	1392	167600	82280	2.03692
22	16564756	72420	56428	1111	147791	72455	2.03973
23	5674615	40392	36650	375	83439	40418	2.06435
24	2616297	26908	22282	150	52649	26941	1.95416
25	4949820	45121	37566	359	90368	45156	2.0012
27	2840742	39429	32668	189	80206	39456	2.03274
29	6392589	45721	38467	389	90634	45753	1.9809
30	5043473	59813	46014	701	120059	59844	2.00617

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
31	12657152	67222	57305	1057	137197	67255	2.03992
35	6867218	47667	41940	522	97732	47704	2.04867
36	15146847	91660	70641	1762	188020	91690	2.05058
38	6468242	60378	52827	629	123394	60412	2.04251
41	8596309	71649	54433	740	141497	71686	1.97382
43	9984139	59455	46833	619	119703	59490	2.01212
44	15511470	64708	55138	961	130823	64748	2.02046

A.6.7 IBRBFS*

Tabla A.10: Resultados del IBRBFS* con el 15-"Puzle"

No.	#h	#F	#p	t	#g	#e	b
00	8468103	11018	14144	136	26735	12675	2.1091
01	222007	2673	3174	5	6067	2959	2.04966
02	1827860	4450	5434	29	10305	5039	2.04464
03	4164980	9100	12401	65	22584	10815	2.08802
04	13927005	15445	18077	233	35444	17208	2.05962
05	1239190	9216	9957	23	20910	10166	2.05665
06	3705037	12515	9615	74	25805	12883	2.00287
07	1895029	6585	6953	34	14296	7190	1.98804
08	18129611	22444	24034	316	50345	24540	2.05146
09	7254922	12971	14552	127	29147	14161	2.05811
10	21245580	41665	34565	505	85585	43068	1.98716
11	20266782	29307	27844	445	62835	30795	2.04036
12	2797349	10488	11699	63	23204	11263	2.06001
13	12851519	21577	21409	248	48076	23173	2.07457
14	4436482	9982	10873	80	21927	11009	1.99155
15	14267214	11884	15115	235	28153	13654	2.06174
16	3626981	9079	9162	67	19670	9636	2.04109

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
17	8236065	22778	22021	165	48509	24093	2.01332
18	12956072	18341	18075	251	39393	19830	1.98644
19	55846031	32410	35129	1028	73560	36056	2.0401
20	151818823	52925	57713	2833	119094	58506	2.03555
23	69362511	29990	37222	1220	71343	34463	2.07007
24	15243090	22886	21814	313	47919	24383	1.96518
25	46390399	36415	37310	900	78561	39436	1.99206
27	185667525	57944	62298	3451	132383	64867	2.04081
28	151580633	52037	58073	2822	120259	58629	2.05115
29	7362086	16001	16461	146	34432	17143	2.0084
30	268916640	73558	72160	5080	159321	78910	2.019
31	104874994	42026	46143	1915	94787	46194	2.05189
32	298636610	67824	73748	5481	152603	74675	2.04353
33	198162795	38145	53527	3239	98236	46843	2.09709
35	42699651	26407	29251	745	59752	29468	2.02762
38	294285930	75746	83842	15490	169124	83097	2.03524
40	67808936	48053	46186	4133	103155	50949	2.02463
41	92123378	57400	54123	4995	123059	61769	1.99221
43	10467282	19618	17957	600	42067	20878	2.0148

A.7 Resultados de las pruebas con grafos

A continuación se muestran los resultados de los mismos algoritmos probados en la Tesis Doctoral, salvo el algoritmo de ramificación y acotación unidireccional o en su versión de perímetro, sobre los problemas expuestos en la tabla 6.8, página 176 de la Tesis Doctoral. En el caso de los algoritmos de perímetro, sólo se han probado las profundidades más favorables: 5 y 10.

A.7.1 Leyenda

En las tablas que se muestran a continuación se han usado los siguientes símbolos en las cabeceras:

No.	Número de instancia.
#h	Número de evaluaciones heurísticas.
#F	Número de actualizaciones de F.
#p	Número de nodos de perímetro generados.
t	Tiempo de ejecución (en segundos).
#g	Número de nodos generados.
#e	Número de nodos expandidos.
b	Factor de ramificación.

El uso de estas abreviaciones mejorará sensiblemente la presentación de las tablas.

A.7.2 IDA*

Tabla A.11: Resultados del IDA* con el problema general de búsqueda en grafos

No.	#h	t	#g	#e	b
$G_{100}^{0,20}$	5049676	207	5049706	2034896	2.48155
$G_{100}^{50,20}$	4	0	6	1	3
$G_{100}^{50,30}$	53553	2	53859	3129	17.2073
$G_{100}^{50,40}$	622598	22	623142	188990	3.2972
$G_{100}^{50,50}$	62	1	1736	19	86.8
$G_{100}^{100,20}$	6	0	6	2	2
$G_{100}^{100,40}$	152145	16	152682	2285	66.79
$G_{100}^{100,50}$	136198	4	136239	39166	3.47841
$G_{150}^{50,20}$	59	0	69	14	4.6
$G_{150}^{50,30}$	20	0	26	5	4.33333

Continúa en la siguiente página →

No.	#h	t	#g	#e	b
$G_{150}^{100,30}$	9	0	12	2	4
$G_{150}^{100,40}$	14	0	16	5	2.66667
$G_{200}^{50,20}$	1811024	415	1813597	233930	7.7527
$G_{200}^{50,40}$	11952	6	17265	17	959.167
$G_{200}^{50,50}$	13083088	553	13083807	3731516	3.5063
$G_{200}^{100,40}$	722	0	729	199	3.645
$G_{250}^{50,30}$	67	0	67	20	3.19048
$G_{250}^{50,40}$	1203979	49	1204610	410218	2.9365
$G_{250}^{100,30}$	8	0	11	3	2.75
$G_{250}^{100,40}$	4870	0	4870	1325	3.6727
$G_{250}^{100,50}$	40	0	45	10	4.09091

A.7.3 RBFS

Tabla A.12: Resultados del RBFS con el problema general de búsqueda en grafos

No.	#h	#F	t	#g	#e	b
$G_{100}^{0,20}$	5049676	0	207	5049706	2034896	2.48155
$G_{100}^{50,20}$	4	0	0	6	1	3
$G_{100}^{50,30}$	53553	0	2	53859	3129	17.2073
$G_{100}^{50,40}$	622598	0	22	623142	188990	3.2972
$G_{100}^{50,50}$	62	0	1	1736	19	86.8
$G_{100}^{100,20}$	6	0	0	6	2	2
$G_{100}^{100,40}$	152145	0	16	152682	2285	66.79
$G_{100}^{100,50}$	136198	0	4	136239	39166	3.47841
$G_{150}^{50,20}$	59	0	0	69	14	4.6
$G_{150}^{50,30}$	20	0	0	26	5	4.33333
$G_{150}^{100,30}$	9	0	0	12	2	4
$G_{150}^{100,40}$	14	0	0	16	5	2.66667

Continúa en la siguiente página →

No.	#h	#F	t	#g	#e	b
$G_{200}^{50,20}$	1811024	0	415	1813597	233930	7.7527
$G_{200}^{50,40}$	11952	0	6	17265	17	959.167
$G_{200}^{50,50}$	13083088	0	553	13083807	3731516	3.5063
$G_{200}^{100,40}$	722	0	0	729	199	3.645
$G_{250}^{50,30}$	67	0	0	67	20	3.19048
$G_{250}^{50,40}$	1203979	0	49	1204610	410218	2.9365
$G_{250}^{100,30}$	8	0	0	11	3	2.75
$G_{250}^{100,40}$	4870	0	0	4870	1325	3.6727
$G_{250}^{100,50}$	40	0	0	45	10	4.09091

A.7.4 BIDA*

Tabla A.13: Resultados del BIDA_5^* con el problema general de búsqueda en grafos

No.	#h	#p	t	#g	#e	b
$G_{100}^{0,20}$	7850096	0	198	2902636	1176886	2.46636
$G_{100}^{50,20}$	172	0	0	6	1	18.8131
$G_{100}^{50,30}$	64546	0	3	53140	2933	17.9895
$G_{100}^{50,40}$	2291791	0	40	623142	188990	3.29722
$G_{100}^{50,50}$	1007	0	0	1736	19	17.6686
$G_{100}^{100,20}$	270	0	0	6	2	17.4466
$G_{100}^{100,40}$	13923	0	1	12450	201	57.8519
$G_{100}^{100,50}$	571400	0	6	93067	27106	3.43348
$G_{150}^{50,20}$	898	0	0	69	14	17.7778
$G_{150}^{50,30}$	538	0	1	26	5	26.197
$G_{150}^{50,40}$	1393338	0	108	1239420	86172	14.3781
$G_{150}^{100,30}$	378	0	2	12	2	35.902
$G_{150}^{100,40}$	345	0	0	16	5	34.566
$G_{200}^{50,20}$	5063229	0	284	1215957	169395	7.17743

Continúa en la siguiente página →

No.	#h	#p	t	#g	#e	b
$G_{200}^{50,40}$	12501	0	7	17262	16	235.31
$G_{200}^{50,50}$	86693884	0	913	13083807	3731516	3.50691
$G_{200}^{100,30}$	6361722	0	4447	6322849	141815	44.5724
$G_{200}^{100,40}$	4526	0	0	729	199	6.27473
$G_{250}^{50,30}$	529	0	0	67	20	8.08989
$G_{250}^{50,40}$	6153913	0	70	1187861	404449	2.93699
$G_{250}^{100,30}$	212	0	26	11	3	162.573
$G_{250}^{100,40}$	18795	0	0	4870	1325	4.11119
$G_{250}^{100,50}$	562	0	6	45	10	83.1379

Tabla A.14: Resultados del BIDA$_1^*$0 con el problema general de búsqueda en grafos

No.	#h	#p	t	#g	#e	b
$G_{100}^{0,20}$	1301879	0	67	319397	132769	2.39789
$G_{100}^{50,20}$	720	0	1	6	1	3.21014
$G_{100}^{50,30}$	16460	0	0	6238	643	7.58613
$G_{100}^{50,40}$	2879271	0	86	623142	188990	3.29595
$G_{100}^{50,50}$	3628	0	2	1736	19	7.24361
$G_{100}^{100,20}$	1032	0	1	6	2	3.06508
$G_{100}^{100,30}$	50908260	0	1363	10757272	3510199	3.06459
$G_{100}^{100,40}$	16933	0	2	12450	201	27.6229
$G_{100}^{100,50}$	633868	0	17	93067	27106	3.41597
$G_{150}^{50,20}$	3817	0	1	69	14	3.7704
$G_{150}^{50,30}$	2568	0	2	26	5	3.49041
$G_{150}^{50,40}$	112441	0	15	91654	2364	29.9322
$G_{150}^{100,30}$	1647	0	4	12	2	3.5659
$G_{150}^{100,40}$	1698	0	1	16	5	3.9902
$G_{200}^{50,20}$	4429301	0	89	453402	98248	4.60775
$G_{200}^{50,40}$	14277	0	10	17262	16	10.133

Continúa en la siguiente página →

No.	#h	#p	t	#g	#e	b
$G_{200}^{50,50}$	168851375	0	3680	13083807	3731516	3.5066
$G_{200}^{100,30}$	406901	0	330	399670	622	229.606
$G_{200}^{100,40}$	13885	0	1	729	199	3.52971
$G_{250}^{50,30}$	1459	0	0	67	20	3.23748
$G_{250}^{50,40}$	10370796	0	134	1187861	404449	2.93735
$G_{250}^{100,30}$	962	0	25	11	3	8.48767
$G_{250}^{100,40}$	31414	0	0	4870	1325	3.79594
$G_{250}^{100,50}$	3044	0	6	45	10	7.8125

A.7.5 RBFPS*

Tabla A.15: Resultados del RBFPS_5^* con el problema general de búsqueda en grafos

No.	#h	#F	#p	t	#g	#e	b
$G_{100}^{0,20}$	4015945	600508	24	117	1486129	600532	2.47467
$G_{100}^{50,20}$	301	0	43	0	6	1	18.8131
$G_{100}^{50,30}$	32734	1404	14	2	27547	1412	19.2206
$G_{100}^{50,40}$	995917	83736	16	20	278909	83737	3.33075
$G_{100}^{50,50}$	10559	5	39	0	1689	6	18.7963
$G_{100}^{100,20}$	180	0	45	1	3	1	17.5171
$G_{100}^{100,40}$	13548	30	10	1	4511	32	96.9574
$G_{100}^{100,50}$	220560	10680	27	3	36281	10687	3.3952
$G_{150}^{50,20}$	423	3	33	0	20	4	19.8548
$G_{150}^{50,30}$	418	1	38	0	10	2	26.6822
$G_{150}^{50,40}$	348383	25103	22	39	276584	25112	11.0021
$G_{150}^{100,30}$	294	0	42	6	6	1	36.0493
$G_{150}^{100,40}$	217	1	31	1	6	2	35.4757
$G_{200}^{50,20}$	2392262	84282	15	110	469900	84285	5.57412
$G_{200}^{50,40}$	212728	4	37	3	5753	5	113.11

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
$G_{200}^{50,50}$	38228397	1655450	37	507	5831659	1655451	3.52407
$G_{200}^{100,30}$	29525	1642	19	48	29297	1645	17.4298
$G_{200}^{100,40}$	1134	35	29	1	131	36	10.1364
$G_{250}^{50,30}$	207	5	15	0	18	6	8.94667
$G_{250}^{50,40}$	1518809	104815	17	25	316121	104817	3.01595
$G_{250}^{100,30}$	315	1	35	26	8	2	163.948
$G_{250}^{100,40}$	8021	592	15	0	2143	593	4.56888
$G_{250}^{100,50}$	340	3	31	6	19	4	87.4364

Tabla A.16: Resultados del RBFPS_1^*0 con el problema general de búsqueda en grafos

No.	#h	#F	#p	t	#g	#e	b
$G_{100}^{0,20}$	520791	52393	68	29	127324	52413	2.40883
$G_{100}^{50,20}$	1260	0	180	1	6	1	3.21014
$G_{100}^{50,30}$	8473	347	62	0	5147	352	9.27534
$G_{100}^{50,40}$	1237917	83736	50	48	278909	83737	3.32751
$G_{100}^{50,50}$	35404	5	146	3	1689	6	7.2514
$G_{100}^{100,20}$	688	0	172	1	3	1	3.06509
$G_{100}^{100,30}$	26055552	1753611	100	761	5346625	1753617	3.04895
$G_{100}^{100,40}$	13968	30	38	1	4511	32	17.1061
$G_{100}^{100,50}$	238388	10680	66	8	36281	10687	3.35915
$G_{150}^{50,20}$	1748	3	150	1	20	4	3.76377
$G_{150}^{50,30}$	2013	1	183	1	10	2	3.48916
$G_{150}^{50,40}$	18240	450	80	2	13752	451	13.123
$G_{150}^{50,50}$	51116350	3547685	65	1470	12056610	3547695	3.39839
$G_{150}^{100,30}$	1281	0	183	4	6	1	3.56561
$G_{150}^{100,40}$	1078	1	154	1	6	2	3.99084
$G_{200}^{50,20}$	2201057	50456	59	46	223740	50457	4.42217
$G_{200}^{50,40}$	891110	4	155	5	5753	5	5.99601

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
$G_{200}^{50,50}$	73875491	1655450	155	1758	5831659	1655451	3.52335
$G_{200}^{100,30}$	12929	44	75	13	12188	45	13.3223
$G_{200}^{100,40}$	3745	35	110	1	131	36	3.51374
$G_{250}^{50,30}$	597	5	45	0	18	6	3.23268
$G_{250}^{50,40}$	2551942	104815	73	40	316121	104817	3.01696
$G_{250}^{100,30}$	1440	1	160	26	8	2	8.48938
$G_{250}^{100,40}$	13483	592	91	1	2143	593	3.83764
$G_{250}^{100,50}$	1820	3	177	6	19	4	7.82332

A.7.6 BRBFS*

Tabla A.17: Resultados del BRBFS* con el problema general de búsqueda en grafos

No.	#h	#F	#p	t	#g	#e	b
$G_{100}^{0,20}$	1485466	35212	10422	10	84750	35484	2.38833
$G_{100}^{50,20}$	7	0	5	0	6	1	3
$G_{100}^{50,30}$	19128	107	2616	0	2858	132	21.4887
$G_{100}^{50,40}$	46124	1283	1478	0	4550	1358	3.34805
$G_{100}^{50,50}$	32921	29	3773	1	3835	38	98.3333
$G_{100}^{100,20}$	4	0	3	0	3	1	1.5
$G_{100}^{100,30}$	52988519	404112	72232	267	1261263	404994	3.11427
$G_{100}^{100,40}$	11044	58	3055	0	3181	73	42.9865
$G_{100}^{100,50}$	89268	1346	2927	1	6616	1398	4.72909
$G_{150}^{50,20}$	23763	12	2090	1	2115	16	124.412
$G_{150}^{50,30}$	36729	1	3071	1	3074	4	614.8
$G_{150}^{50,40}$	58006	156	12588	2	12927	182	70.6393
$G_{150}^{100,20}$	25119092	200946	87107	132	695539	201569	3.45061
$G_{150}^{100,30}$	7	0	6	0	6	1	3
$G_{150}^{100,40}$	26749	0	3350	1	3351	3	837.75

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
$G_{200}^{50,20}$	1097210	19667	9898	7	64400	19966	3.22532
$G_{200}^{50,40}$	13250454	8	12652	33	12671	13	905.071
$G_{200}^{100,30}$	12172	4	12076	10	12090	9	1209
$G_{200}^{100,40}$	231239	154	11759	2	12064	188	63.8307
$G_{250}^{50,30}$	22049	16	2243	0	2273	25	87.4231
$G_{250}^{50,40}$	418261	9752	4707	3	28960	9955	2.9088
$G_{250}^{50,50}$	29238027	198471	82703	199	830028	199086	4.16917
$G_{250}^{100,30}$	56005	2	18669	27	18672	3	4668
$G_{250}^{100,40}$	604840	1195	22912	4	25900	1268	20.4098
$G_{250}^{100,50}$	279575	9	27969	20	27989	14	1865.93

A.7.7 IBRBFS*

Tabla A.18: Resultados del IBRBFS* con el problema general de búsqueda en grafos

No.	#h	#F	#p	t	#g	#e	b
$G_{100}^{0,20}$	23273	2	3540	0	690	322	2.13622
$G_{100}^{0,30}$	224713	21	34010	1	2768	1218	2.27071
$G_{100}^{0,40}$	124862	9	23262	1	1826	702	2.59744
$G_{100}^{0,50}$	2303104	660	195948	16	13793	7413	1.8604
$G_{100}^{50,20}$	7	0	5	0	6	1	3
$G_{100}^{50,30}$	13329	0	2667	1	446	43	10.1364
$G_{100}^{50,40}$	4829	0	1031	0	263	79	3.2875
$G_{100}^{50,50}$	31479	0	6752	1	2579	158	16.2201
$G_{100}^{100,20}$	4	0	4	0	3	1	1.5
$G_{100}^{100,30}$	192924	158	15417	1	3777	1075	3.51022
$G_{100}^{100,40}$	23885	0	3694	0	1527	18	80.3684
$G_{100}^{100,50}$	27441	0	3214	0	532	104	5.06667
$G_{150}^{0,20}$	359152	52	42659	3	4160	2111	1.9697

Continúa en la siguiente página →

No.	#h	#F	#p	t	#g	#e	b
$G_{150}^{0,30}$	888628	224	100032	7	7157	3534	2.02461
$G_{150}^{0,40}$	26894603	19757	1150466	221	119812	67173	1.78361
$G_{150}^{50,20}$	10446	0	2184	0	1058	8	117.556
$G_{150}^{50,30}$	18369	1	3068	1	3068	3	767
$G_{150}^{50,40}$	63356	0	12033	1	2338	91	25.413
$G_{150}^{50,50}$	1894551	21	203984	16	14111	2879	4.89965
$G_{150}^{100,20}$	318312	206	21276	1	3627	1381	2.62446
$G_{150}^{100,30}$	7	0	7	0	6	1	3
$G_{150}^{100,40}$	13377	0	3350	0	3347	2	1115.67
$G_{150}^{100,50}$	567291	0	72333	5	5422	1776	3.05121
$G_{200}^{0,20}$	2046421	1025	223223	17	14592	9249	1.57751
$G_{200}^{0,30}$	4370941	679	400929	39	20649	10746	1.92137
$G_{200}^{50,20}$	50340	58	5445	0	1451	544	2.66239
$G_{200}^{50,40}$	11930	3	4706	0	2519	70	35.4789
$G_{200}^{50,50}$	4782405	5	865519	111	29678	12574	2.36008
$G_{200}^{100,20}$	10348731	6561	427434	55	71751	31652	2.2668
$G_{200}^{100,30}$	84484	0	12076	10	12078	7	1509.75
$G_{200}^{100,40}$	38382	1	9679	0	953	56	16.7193
$G_{250}^{0,20}$	2719615	433	239551	20	14220	7076	2.00933
$G_{250}^{0,30}$	19744240	13064	1167088	222	84408	47412	1.78027
$G_{250}^{50,20}$	1479123	20	169750	11	3783	1063	3.55545
$G_{250}^{50,30}$	8674	1	1986	0	821	114	7.13913
$G_{250}^{50,40}$	188736	622	14001	8	8125	957	8.48121
$G_{250}^{50,50}$	253665	12	23071	2	2320	585	3.95904
$G_{250}^{100,20}$	3939395	0	568383	57	16303	5046	3.23024
$G_{250}^{100,30}$	56002	0	18667	31	18669	2	6223
$G_{250}^{100,40}$	59522	1	13632	1	978	113	8.57895
$G_{250}^{100,50}$	121351	2	20802	7	9357	13	668.357