UNIVERSIDAD NACIONAL DE LA PLATA (UNLP)

FACULTAD DE INFORMÁTICA

Tesina de Licenciatura en Informática

Paralelización de problemas de búsqueda en grafos en una arquitectura tipo clúster. Análisis de performance.

Autor: Victoria María Sanz

Director: Ing. Armando E. De Giusti

Codirector: Dr. Marcelo Naiouf

Febrero de 2009

Índice

Resumen

Los problemas de optimización discreta, también conocidos como problemas combinatorios, surgen en diversas áreas y en general se resuelven utilizando técnicas que buscan una solución en el espacio de estados implícito del problema.

Debido a la alta complejidad computacional de esta clase de problemas, las búsquedas exhaustivas se vuelven inaceptables, por lo cual se han desarrollado algoritmos heurísticos que utilizan funciones para evaluar el costo de los nodos y de este modo procesar primero los nodos que se estima están más cercanos al nodo "solución óptima".

Es de interés el desarrollo de heurísticas más potentes y algoritmos paralelos que resuelvan los problemas de optimización discreta de forma eficiente, con el fin de resolver instancias cada vez más grandes y dado que algunos problemas requieren soluciones en tiempo real, el paralelismo es en muchos casos la única forma de obtener el tiempo de respuesta esperado.

En este marco, este trabajo toma como caso de estudio un problema de optimización clásico llamado Puzzle N^2-1, y presenta una solución secuencial basada en el algoritmo A*. Se estudian variantes de la función heurística clásica (basadas en la Distancia de Manhattan) y se expone un trabajo experimental para analizar las mejoras en el rendimiento producidas, partiendo de diferentes configuraciones iniciales.

Se propone una solución paralela al problema del Puzzle N^2 -1 sobre una arquitectura tipo cluster, y se analiza el speedup, la eficiencia, y la superlinealidad a medida que se escala el número de procesadores y el tamaño del problema (N).

Se presenta además una generalización del problema para su aplicación a la planificación de movimientos de robots con múltiples objetivos.

Objetivos

Investigar y estudiar la performance de los algoritmos paralelos de búsqueda en grafos sobre un cluster con diferentes configuraciones, tomando como caso de estudio problema del Puzzle N^2-1, así como el análisis que surge por la aplicación de distintas heurísticas para la evaluación de los nodos durante la búsqueda. Se plantea además una generalización del problema para su aplicación a problemas de movimientos de robots con múltiples objetivos.

Motivación

Los algoritmos de búsqueda sobre un espacio de estados pueden ser aplicados para resolver problemas combinatorios. El propósito de dichos algoritmos es encontrar una solución óptima para el problema planteado.

La gran demanda de recursos que surge cuando el espacio de estados se torna exponencial hace imprescindible el desarrollo de soluciones paralelas para esta clase de problemas.

Un problema de optimización clásico es el Puzzle N^2-1, una generalización del problema Puzzle 15 propuesto por Sam Lloyd. Dicho problema es de especial interés en el área de Inteligencia Artificial, ya que ha sido usado para pruebas en búsquedas heurísticas, teniendo además aplicaciones en el campo de robótica, debido a que estos algoritmos encuentran un plan para alcanzar un estado objetivo desde un estado inicial en la menor cantidad de pasos posible.

Se plantea una solución secuencial, basada en el algoritmo A*. Se estudiarán variantes de la heurística clásica (basadas en la Distancia de Manhattan) y se realizará un trabajo experimental para analizar las mejoras en la performance producidas, partiendo de diferentes configuraciones iniciales.

Se propone una solución paralela al problema del Puzzle N^2-1 sobre una arquitectura tipo clúster, y se analizará el speedup, la eficiencia, y la posible superlinealidad a medida que se escala el número de procesadores y el tamaño del problema (N).

Se presenta una generalización del problema para su aplicación a la planificación de movimientos de robots con múltiples objetivos.

Estudio y desarrollo a realizar

- ✓ Presentar una solución secuencial para el problema del Puzzle N^2-1.
- ✓ Presentar una solución paralela para el problema del Puzzle N^2-1 sobre un clúster.
- ✓ Desarrollar variaciones de la heurística clásica (Distancia de Manhattan), a ser utilizadas tanto en el algoritmo secuencial como el algoritmo paralelo.
- ✓ Realizar un trabajo experimental con las diferentes heurísticas y evaluar la mejora en tiempo y nodos procesados al utilizar heurísticas más afinadas.
- ✓ Analizar la performance obtenida al paralelizar (speedup, eficiencia, superlinealidad). En particular es interesante investigar qué factores causan la superlinealidad, y también estudiar qué ocurre al aumentar el volumen de trabajo a realizar (escalar el problema) y al aumentar la cantidad de procesadores para su resolución.
- ✓ Realizar una generalización del problema del Puzzle N^2-1 para su aplicación en robótica.

Trabajos presentados sobre el tema

Sobre esta temática he presentado trabajos en distintos congresos, citados a continuación:

- ✓ *Superlinearidad sobre Clusters. Análisis experimental en el problema del Puzzle N^2 -1*. CACIC 2007 (XIII Congreso argentino de ciencias de la computación).
- ✓ *Resolución paralela del problema Puzzle N^2-1 sobre un cluster.* XV Jornadas de Jóvenes Investigadores de la Asociación de Universidades Grupo Montevideo.
- ✓ *Parallel Processing Puzzle N^2-1 on Cluster Architectures. Performance Analysis.* ITI 30th International Conference on Information Technology Interfaces.
- ✓ *Análisis de Performance en el procesamiento paralelo sobre Clusters del problema del Puzzle N^2-1.* ENIEF 2008. XVII Congreso sobre Métodos Numéricos y sus Aplicaciones.

1. Introducción

La evolución hacia el procesamiento paralelo para acelerar la resolución de tareas ha sido evidente en las últimas décadas. Entre los factores que conducen a la realización de software concurrente y hardware para multiprocesamiento se destacan:

- ✓ El procesamiento de información para la toma de decisiones en tiempo real en ambientes industriales y administrativos (robótica, industria, sistemas multimedia en tiempo real, reconocimiento de patrones).
- ✓ La necesidad de reducir el tiempo de procesamiento de grandes volúmenes de datos.

El crecimiento del poder de cómputo debido a la evolución de la tecnología usada en los componentes y arquitecturas paralelas (supercomputadoras, hipercubos de procesadores homogéneos, grandes redes de procesadores heterogéneos, procesadores especializados en imágenes, procesadores para el tratamiento de señales) da la posibilidad de resolver problemas científicos complejos tales como modelos de sistemas biológicos, reacciones químicas, física de partículas, de ingeniería, predicciones de incendios y meteorología que de otra forma no tendrían solución en tiempo adecuado.

Por otra parte, en la actualidad se han vuelto habituales entre los usuarios las computadoras de escritorio con dos, cuatro e incluso ocho procesadores y las computadoras *stand-alone*[1] conectadas en red formando un cluster. Estas arquitecturas son más económicas que las supercomputadoras, por lo que el estudio de la posible performance a obtener en los algoritmos sobre las mismas ha adquirido importancia.

Una de las áreas de gran interés en el cómputo paralelo en los últimos años es el procesamiento paralelo de búsquedas en grafos.

Los problemas de optimización discreta (DOP) abarcan un gran número de áreas [SER06] y a menudo son resueltos con algoritmos de búsqueda en grafos que exploran el espacio de estados del problema buscando un estado "solución" que minimice una función objetivo [LAM04].

En general estas técnicas de búsqueda tienen alto costo computacional debido a que los espacios de estados crecen en orden factorial o exponencial. En muchos casos es imposible el análisis exhaustivo del espacio de soluciones, de modo que debe recurrirse a algoritmos que utilicen heurísticas para estimar el costo de los estados y procesar primero los nodos más prometedores. [GRA99] [PAR95]

La alta complejidad computacional, tanto en tiempo de cómputo y en cantidad de memoria utilizada, impulsan el desarrollo de algoritmos paralelos para los problemas de optimización discreta de modo de resolverlos eficientemente, y en particular las técnicas

[1] El término "Stand alone" se refiere a que cada computadora puede ser utilizada en forma independiente, ya que posee su propio hardware y sistema operativo.

de procesamiento de grafos que representen el problema han sido de gran interés. [FER96] [REI93]

Este es el caso de algunas variantes del método de búsqueda BFS (Best First Search) que parten de un nodo del grafo que representa el problema a resolver y utilizan alguna métrica de estimación del trabajo para alcanzar la solución, de modo de evolucionar a partir del estado inicial del grafo hacia el estado "solución óptima".

La paralelización natural de la técnica consiste en iniciar la evolución de diferentes nodos "posibles" sobre los distintos procesadores de la arquitectura multiprocesador. A medida que el algoritmo progresa es necesario comunicar los procesadores para informar resultados parciales alcanzados y posibilitar la detección de terminación de la búsqueda o bien descartar soluciones, de acuerdo a la métrica elegida, que no mejorarán la solución parcial encontrada hasta el momento. [HAN92] [KOR05]

Es interesante reflexionar sobre algunos aspectos que se dan al utilizar arquitecturas paralelas tipo clúster en la resolución de problemas de optimización discreta [AND95]:

- ✓ La *granularidad* de la paralelización es crítica, porque de ella dependerá la mejora en el tiempo de solución y también el overhead de comunicaciones.
- ✓ En general el *balance de carga* tiene que ser dinámico, ya que el espacio de estados es implícito y generado durante la búsqueda. Esto exige comunicación ya que el trabajo exploratorio es variable y es muy difícil predecirlo a priori. [BOH02]

En procesamiento paralelo uno de los puntos principales de interés en la resolución de un algoritmo sobre una arquitectura multiprocesador es el *factor de Speedup Sp*. Dicho factor es una medida de perfomance relativa definida como la relación entre el tiempo de ejecución del mejor algoritmo secuencial sobre una máquina monoprocesador y el tiempo de ejecución del algoritmo paralelo sobre una máquina multiprocesador (Grama et al, 2003; Leopold, 2001). Si llamamos Ts al tiempo de ejecución secuencial y Tp al paralelo, tenemos la relación ***Sp=Ts/Tp*** que normalmente se trata de maximizar en el desarrollo de aplicaciones paralelas. Sp está limitado por el máximo grado de concurrencia que puede obtenerse de la aplicación, por el inevitable componente secuencial del algoritmo y por el número de procesadores N disponibles para la ejecución. [QUI93]

Un segundo parámetro de importancia al analizar aplicaciones paralelas es la *Eficiencia E* alcanzada. Se define como Eficiencia la relación entre el Speedup y el número de procesadores utilizados para obtenerlo: ***E=Sp/N***. Esta definición pone la Eficiencia entre 0 y 1. Alcanzar valores cercanos a 1 significa que se logra Sp cercano al óptimo N. *E* resulta una métrica de calidad y de costo del algoritmo paralelo que es particularmente importante y no siempre se puede mantener al escalar los problemas, al incrementar el número de procesadores o al portar el algoritmo sobre otra arquitectura multiprocesador. [BUY99]

La *Escalabilidad* es un factor muy significativo en las aplicaciones paralelas: normalmente los problemas "escalan", es decir aumenta el volumen de trabajo a realizar, y también las arquitecturas multiprocesador que utilizamos pueden "escalar" incrementando los procesadores utilizados. Es de interés investigar el efecto de escalar trabajo y/o procesadores sobre el rendimiento de los algoritmos paralelos, considerando *Sp* y *E*. [HWA93]

El máximo Speedup teórico puede en algunos casos ser mejorado y esto da lugar al concepto de *Superlinealidad Su*. Es interesante analizar por qué Sp puede superar N, en particular en los problemas de optimización discreta:

- ✓ La exploración del espacio total de soluciones posibles puede reducirse al distribuir el trabajo entre N procesadores y poder "cortar" o "finalizar" la búsqueda global al llegar al resultado esperado en cualquiera de ellos [HEL90] [MAN02]. Es decir que en teoría la arquitectura de clúster podrá permitirnos alcanzar superlinealidad, dependiendo del balance de carga, la heterogeneidad de los procesadores y la relación tiempo de procesamiento/tiempo de comunicaciones del algoritmo empleado. [SAN07]

- ✓ Si utilizamos arquitecturas distribuidas aún más débilmente acopladas (como multiclusters o Grid), la relación entre tiempo de procesamiento y tiempo de comunicación marcará un límite a la posibilidad de alcanzar superlinealidad [WIL05].

Este trabajo investiga las ventajas de utilizar un cluster de computadoras para resolver un problema combinatorio. El caso de estudio seleccionado es el problema del Puzzle N^2-1, un problema de optimización discreta clásico, con aplicaciones en el área de robótica e Inteligencia artificial, el cual es generalmente usado como benchmark para estimar el rendimiento de un sistema paralelo en la resolución de algoritmos heurísticos de búsqueda.

2. Caracterización del problema

Los problemas de optimización discreta (DOP), también conocidos como problemas combinatorios, se basan en encontrar una solución que cumpla con las restricciones del mismo y que minimice una función objetivo, es decir la solución buscada es la óptima.

Diversos problemas de este tipo han surgido en distintos campos, tales como el diseño de circuitos integrados a gran escala (VLSI), selección de rutas óptimas, administración de tareas para ser realizadas en tiempo óptimo, entre otras.

En general los DOP son problemas NP complejo, por lo que han ganado gran interés debido a su alto costo computacional, tanto en tiempo de procesamiento y cantidad de memoria utilizada, razón por la cual se han estudiado técnicas para resolverlos eficientemente: algoritmos heurísticos, funciones heurísticas más potentes, y en especial algoritmos paralelos de búsqueda.

Para algunos problemas, como planeamiento de movimientos de robots y scheduling de tareas, se requieren soluciones en tiempo real, por lo que el procesamiento paralelo es la única forma de obtener un rendimiento aceptable.

A continuación, se define formalmente qué es un problema de optimización discreta, pasando luego a describir el problema tomado como caso de estudio y al final de la sección se plantean problemáticas reales en los que se aplican los DOP.

2.1 Problema de Optimización Discreta (DOP)

Un problema de optimización discreta se define de la siguiente forma:

> *Sea **S** un conjunto finito o infinito contable de soluciones que satisfacen las restricciones del problema, y **f** una función que aplica un costo a cada elemento de S (f: S => R), resolver un DOP implicará encontrar un* $x^* \in S$ *tal que* $f(x^*) \leq f(x)$, *para todo* $x \in S$.

En la mayoría de los problemas el conjunto S es bastante grande, y su enumeración exhaustiva para encontrar la solución óptima se torna imposible. Por este motivo, el problema se puede replantear como una búsqueda de un camino de costo mínimo en un grafo, comenzando a partir de un nodo inicial (problema a resolver), y finalizando en un nodo solución. Un elemento de S puede ser visto como un camino entre el nodo inicial y final del espacio de estados.

El grafo recibe el nombre de *espacio de estados*, y a cada nodo del grafo se lo llama *estado*.

2.2 Caso de estudio: Puzzle N^2-1

2.2.1 Interés y aplicaciones

El problema propuesto como caso de estudio es un problema de optimización discreta NP complejo [RAT86]. En general, los algoritmos de resolución paralela para el mismo permiten alcanzar un Speedup mayor al teóricamente posible –superlinealidad- [SAN07]. Es de interés descubrir para el problema del Puzzle cuáles son los factores que llevan a dichos resultados.

Por otra parte, el Puzzle es un problema dependiente de los datos, ya que los tiempos para resolver una entrada de tamaño N varían considerablemente entre sí. Esto se debe a que dicho tiempo depende del grado de desorden inicial del tablero a resolver, de la función de estimación a utilizar durante la búsqueda, entre otros factores. Evaluar cuáles son las variables que influyen sobre el rendimiento es de especial interés, ya que todo aporte a la familia de problemas no deterministas es un avance debido a la falta de información en dicha área.

Es importante destacar también las áreas de aplicación del problema. En el campo de la robótica son comunes los algoritmos que permitan buscar planes óptimos para aplicarlos a movimientos de robots [FIT05]. En Inteligencia Artificial este tipo de juegos se utilizan para diseñar y testear algoritmos heurísticos.

Asimismo, dada la simplicidad del problema, el mismo es usado generalmente como benchmark[2] para estimar el rendimiento de un sistema paralelo en la resolución de algoritmos heurísticos de búsqueda.

2.2.2 Definición del problema

El problema del Puzzle N^2-1 es una generalización del Puzzle-15 ideado por Sam Lloyd [RAT90]. Consiste en N^2-1 piezas numeradas de 1 a N^2-1 colocadas en un tablero de tamaño N^2. N^2-1 casilleros del tablero contienen exactamente una pieza, quedando sólo una casilla vacía la cual se denomina “hueco”.

Un movimiento legal en este juego implica mover el hueco a una posición adyacente a él, en sentido horizontal o vertical, trasladando la ficha que estaba en ese lugar a la posición anterior del hueco.

El objetivo del Puzzle es aplicar movimientos legales repetidamente hasta convertir el tablero inicial en el tablero final elegido. Un tablero final clásico es aquel donde en la

[2] Benchmark hace referencia a la acción de ejecutar una aplicación para evaluar el rendimiento relativo de un objeto, para así poder comparar diferentes arquitecturas o sistemas.

casilla (i,j) se encuentra la pieza numerada como (i-1)*N + j y en la casilla (N,N) se encuentra el hueco.

La figura 2.1.a muestra un Puzzle N^2-1 donde N es 4. La figura 2.1.b representa el tablero solución clásico.

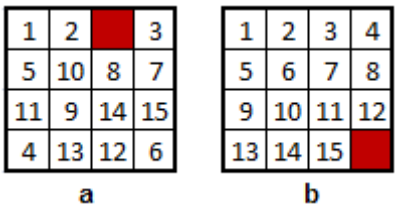

Figura 2.1: Tableros de Puzzle 15 (N=4)
a. Tablero inicial. **b.** Tablero final.

La solución al problema planteado tendrá que ser aquella que minimice la cantidad de movimientos que deben realizarse para alcanzar la configuración final desde la configuración inicial dada.

2.2.3 Solubilidad

El problema original del Puzzle-15, planteado por Sam Lloyd, consistía en encontrar la secuencia de movimientos que transforme el tablero de la figura 2.2.a en el tablero solución clásico, e incluso ofreció una recompensa. No obstante, nadie pudo resolver dicho problema y esto se debe a que no existe solución para el mismo [JOH79].

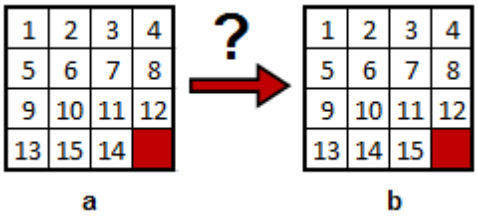

Figura 2.2: Problema original planteado por Sam Lloyd.

El espacio de estados del Puzzle N^2-1 tiene N^2! estados. Este número surge ya que un tablero puede representarse como un vector de N^2 posiciones, y en cada posición debe colocarse una ficha o el hueco, por lo tanto cualquier ordenamiento de los mismos es admitido como un tablero.

Sin embargo, no todo estado puede alcanzarse desde otro aplicando movimientos legales. Esto se debe a que el grafo que representa el espacio de estados del Puzzle tiene dos componentes conexas de igual tamaño, por lo tanto si el estado inicial y el final no están en la misma componente entonces no hay solución para el problema.

En la figura 2.3.a se ilustra la situación en la que el tablero inicial y final están en la misma componente conexa, por lo que existe al menos una solución para el problema.

En la figura 2.3.b se muestra un caso donde el estado final es inalcanzable desde el estado inicial (no hay solución).

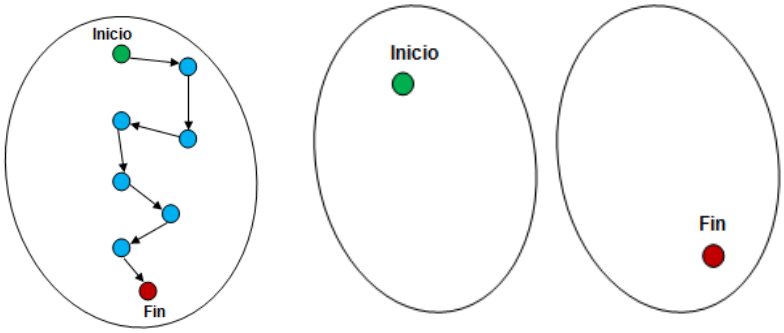

Figura 2.3: Espacio de estados para el problema del Puzzle
a. Estado inicial soluble para el estado final **b.** Estado inicial no soluble para el estado final

A partir de lo comentado anteriormente, se puede afirmar lo siguiente: sólo la mitad de los estados es alcanzable desde cualquier otro. Aplicando un simple paso de detección previo a la búsqueda, para determinar si la configuración final es alcanzable desde la configuración inicial, se reduce el espacio de estados del Puzzle N^2-1 a N^2!/2.

Algoritmo para la detección de solubilidad:

El procedimiento para verificar si un tablero inicial tiene solución para un tablero final es el siguiente:

- ✓ Para cada ficha i (i = 2..N^2-1), se cuenta la cantidad de piezas con menor número que aparecen después de ella, ya sea en la misma fila a su derecha, o en cualquier fila inferior. Llamaremos a este número "inversión de i", y se denotará n_i.
- ✓ A continuación se calcula para el tablero inicial y final NT = $n_2 + n_3 + \ldots n_{(N^2-1)}$ (la suma de las inversiones de todas las fichas). Si N es par, se le suma a NT el número de la fila donde se encuentra el hueco. [3]
- ✓ Si la paridad de ambos resultados es la misma, entonces el tablero final es alcanzable desde el tablero inicial. Esto se debe a que (NT mod 2) se mantiene invariante con cualquier movimiento legal.

En la figura 2.4 se muestra un ejemplo para N par, donde el tablero 2.4.a tiene *NT*=28 y para el tablero 2.4.b. *NT*=4, por lo tanto el tablero de la figura 2.4.a tiene solución.

[3] Las filas se cuentan desde arriba hacia abajo, comenzando desde 1.

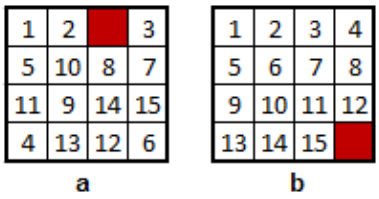

Figura 2.4: Tableros de Puzzle 15 (N=4). Ejemplo para N par
a. Tablero inicial. **b.** Tablero final.

En la figura 2.5 se muestra un ejemplo para N impar, donde el tablero 2.5.a tiene *NT*=50 y para el tablero 5.b. *NT*=0, por lo tanto el tablero de la figura 2.5.a tiene solución.

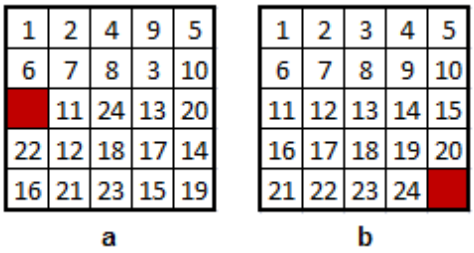

Figura 2.5: Tableros de Puzzle 15 (N=5). Ejemplo para N impar
a. Tablero inicial. **b.** Tablero final.

2.3 Otras clases de problemas similares y aplicaciones

Las búsquedas en espacios de estados son comunes en diversas áreas. Un ejemplo es la asignación de celulares a antenas. Al encender un celular este es asignado a una antena cercana y a una radiofrecuencia para comunicarlo con la antena. Hay millones de celulares y un número limitado de antenas y frecuencias disponibles, por lo que el sistema de telefonía celular tiene que realizar estas asignaciones lo más económicamente posible.

Algo similar ocurre en la planificación de vuelos de las aerolíneas. Las aerolíneas asignan su personal a los vuelos y para ello deben tener en cuenta planes de vuelo, regulaciones e seguridad, regulaciones de trabajo, etc. Esta asignación debe ser realizada de forma óptima para obtener ganancias.

Otros problemas de optimización discreta clásicos son:

- ✓ El *problema del viajante de comercio* (TSP), en donde se tiene un conjunto de ciudades y se debe encontrar una ruta que recorra todas las ciudades con el menor costo posible.

- ✓ El *problema de la mochila discreta*, que consta de una mochila en la que de deben colocar objetos. Cada objeto tiene una utilidad y ocupa determinado lugar en la mochila. El problema surge cuando se debe elegir que objetos seleccionar de forma de obtener el máximo beneficio (tener todo lo que se necesita) sin exceder la capacidad de la mochila. [MAR90]

- ✓ En la *planificación de movimientos de robot* se busca un plan óptimo para trasladar al robot desde un punto a otro, sin colisionar con obstáculos si los hubiese. También hay casos donde el robot dispone de un brazo mecánico de N componentes y se debe encontrar la secuencia de movimientos óptima de las partes para alcanzar un punto en el espacio donde se encuentra el objeto a manipular. [GER05]

3. Algoritmos de búsqueda en espacios de estados

Los algoritmos de búsqueda en grafos pueden ser usados para resolver problemas de optimización discreta. Dichas técnicas de búsqueda se diferencian de los métodos convencionales en que el grafo que representa el espacio de estados es implícito, es decir los nodos se generan a medida que la búsqueda avanza.

Estos algoritmos se dividen en dos categorías: Depth-First y Best-First, y resuelven instancias procesando repetidamente estados hasta encontrar un estado "solución", el cuál podría ser óptimo o no, dependiendo de la variante utilizada. Muchos algoritmos de búsqueda podrán determinar la solución óptima buscando sólo en una porción del grafo.

A continuación se describe brevemente los elementos para construir el espacio de estados del problema, pasando luego a clasificar las técnicas de búsqueda, para concluir con un análisis de los algoritmos de búsqueda y la elección de aquel que se usará para resolver el problema propuesto como caso de estudio.

3.1 Construcción del espacio de estados del problema

Para construir el grafo sobre el cual se realizará la búsqueda se requiere:

- ✓ *Estado inicial:* estado desde el que se comienza la búsqueda. Representa el problema a resolver.
- ✓ *Estado final:* es una configuración que se desea alcanzar. El estado final puede ser único (por ejemplo: el tablero final clásico en el Puzzle) o pueden haber varios estados meta (por ejemplo: alguno de los tableros donde las fichas se ordenan en forma ascendente, y el hueco se ubique en cualquiera de las cuatro esquinas). En este último caso se busca un estado que cumpla una condición.
- ✓ *Esquema de ramificación:* utilizado para generar subproblemas a partir de un problema dado. En el caso del Puzzle, dado un tablero se obtienen subproblemas al mover el hueco a sus posiciones adyacentes, obteniendo así a lo sumo 4 subtableros.
- ✓ *Estrategia de búsqueda:* estrategia para seleccionar un nodo entre los nodos pendientes de acuerdo a prioridades definidas. Generalmente la estrategia se basa en seleccionar nodos que estén próximos a la solución (Best-First) o el nodo generado más recientemente (Depth-First).
- ✓ *Función de costo:* estima el costo de alcanzar la solución a partir de un nodo intermedio (heurística h(n)) o el costo del camino actual desde el nodo inicial al nodo solución pasando por el nodo que se está valuando. Siendo n el nodo a valuar, dicha función se define como $f(n) = g(n) + h(n)$.

Las funciones de costo son utilizadas sólo por algunas variantes de las técnicas Depth-First y Best-First.

3.2 Clasificación BFS y DFS general

El método simple de búsqueda **Depth-First *(DFS)*** comienza a evolucionar nodos desde un estado inicial. En cada iteración se selecciona uno de los nodos más recientemente generados para su procesamiento y ramificación. Cuando se encuentra una solución el algoritmo termina, no garantizando que la misma sea óptima. Si se busca la mejor solución, este algoritmo procesaría el grafo completo.

En los ejemplos se supone el grafo de la figura 3.1.

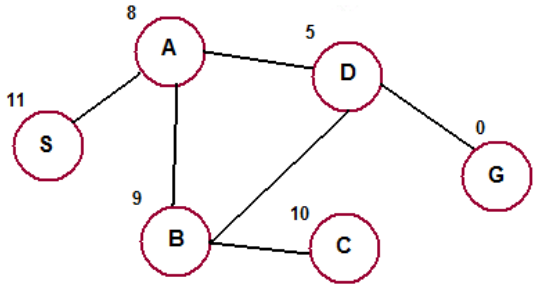

Figura 3.1: Grafo ejemplo.

En la figura 3.2 se visualiza el espacio de estados generado durante la búsqueda, donde los nodos son procesados en el orden indicado por su numeración.

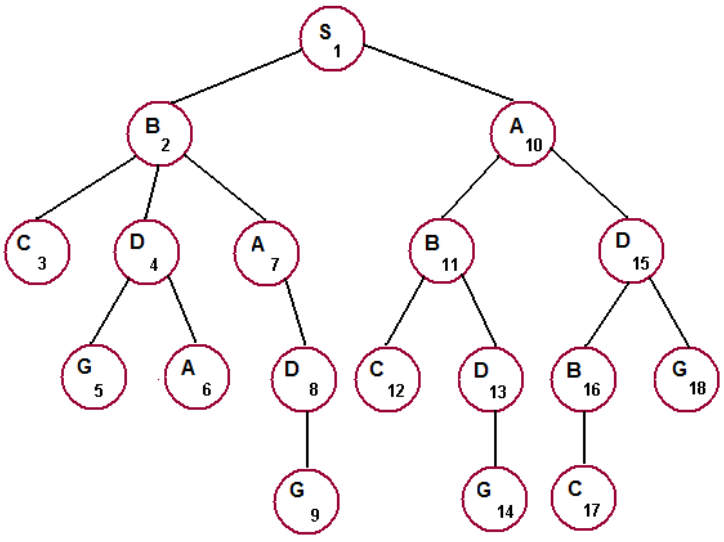

Figura 3.2. Recorrido de un grafo por el método simple DFS.

La ventaja de esta técnica es que utiliza espacio de memoria lineal ya que sólo almacena la rama siendo explorada.

La desventaja ocurre cuando el nodo solución, aún estando cercano a la raíz, demora en ser encontrado debido a que se está procesando una rama muy profunda. En estos casos sería útil incluir un límite para que, al ser alcanzado, se dejara de procesar esa rama. Si con ese límite no se encontró ninguna solución, el límite es actualizado y se realiza la búsqueda nuevamente.

Las distintas variantes se distinguen en la forma de tomar el nodo sucesor a procesar próximamente o en el tipo de solución que encuentran:

- ✓ *Ordered Backtracking (Hill Climbing):* los sucesores de un nodo son ordenados según su cercanía de la solución (estimada).

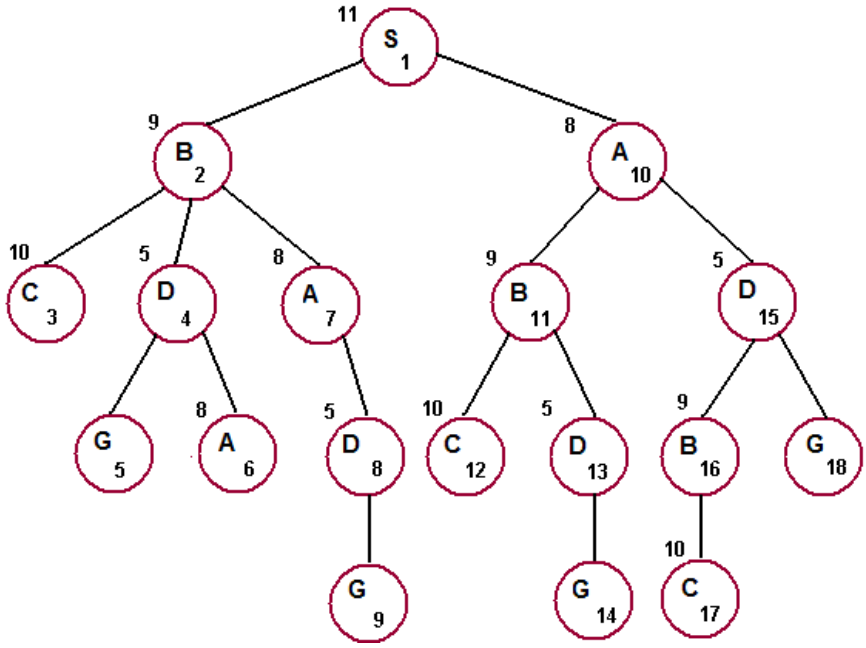

Figura 3.3: Recorrido del grafo por la variante Ordered Backtracking.

La figura 3.3. muestra el árbol que se genera implícitamente durante la búsqueda a partir del grafo de la figura 3.1. Arriba de cada nodo del grafo se muestra su estimación heurística.

Al principio, se procesa nodo de comienzo S, el cual es expandido generando los nodos 2, con estimación 9, y 10 con estimación 8. El nodo siguiente a procesar será el 10, que se estima está más cercano a la solución. Al expandir el 10, se generan los nodos 11 (con estimación 9) y 15 con estimación 5. Como el 15 es el sucesor con menor costo, pasa a procesarse. Así continua la búsqueda hasta encontrar el nodo 18, que es la solución.

- ✓ *Iterative Deepening (Depth-bounded DFS):* DFS en el que se incluye un límite correspondiente a la profundidad de la rama actual. Si al procesar un nodo la expansión del mismo hará que la profundidad de la rama sobrepase el límite, el algoritmo no expande dicho nodo y vuelve atrás, pasando a

expandir el próximo nodo generado más recientemente. Esta variante no garantiza encontrar la mejor solución.

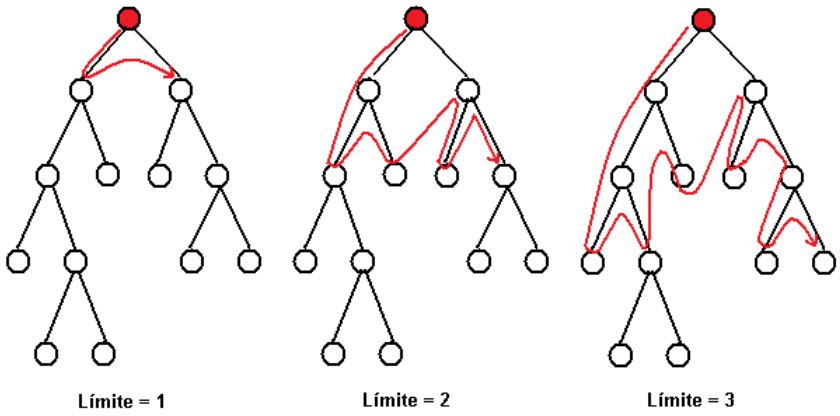

Figura 3.4. Etapas en la búsqueda con el método Iterative Deepening

- ✓ *IDA*:* DFS en el que se incluye un límite en términos de costo de cercanía a la solución. Al principio, el límite se establece con el valor del costo estimado de alcanzar la solución a partir del nodo inicial (h(n)). Si la solución no es encontrada dentro de ese límite, en la próxima iteración se establece el límite con el valor f(n) [4] del nodo cuyo valor era el menor que traspasó el límite. En caso de encontrar la solución habiendo buscado en toda la frontera actual, dicha solución es la óptima.

- ✓ *Depth-First Branch and Bound (DFBB):* esta variante garantiza encontrar la mejor solución. Básicamente, realiza un DFS simple valuando los nodos, pero además mantiene el costo de la solución mínima encontrada hasta el momento. Al encontrar una nueva solución, si es mejor que la actual, la actualiza. Aquellas ramas que no alcanzarán una solución mejor que la actual son podados.

  La figura 3.5 muestra una representación del grafo, destacando el nodo actual a procesar y su subárbol. Si este nodo tiene costo f(x) y el costo de la mejor solución encontrada hasta el momento es f(s), si f(x) >= f(s) entonces el nodo es podado, es decir su subárbol no será procesado ya que no llevará a una solución mejor.

[4] La función f(n) se define f(n) = h(n) + g(n), donde g(n) es la distancia desde el nodo inicial al nodo n.

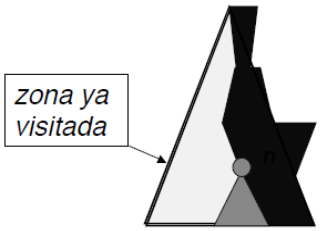

Figura 3.5. Grafo de estados y nodo actual, según su costo se descarta el subgrafo.

Todas las variantes de la técnica de búsqueda **Best-First (BFS)** valuan los nodos mediante una función de costo. De este modo, mantienen una lista abierta, con los nodos no procesados, y en cada iteración procesan el nodo más prometedor disponible en dicha estructura, ramificándolo e insertando los nodos adyacentes en la lista abierta según ciertas condiciones, o actualizando su costo. Además soportan el chequeo de ciclos manteniendo una lista cerrada, con los nodos ya procesados.

Estas variantes difieren en la función de costo a utilizar:

- ✓ *Breadth First Search:* toma como función de costo la altura del nodo. Encuentra la solución más cercana al nodo raíz.

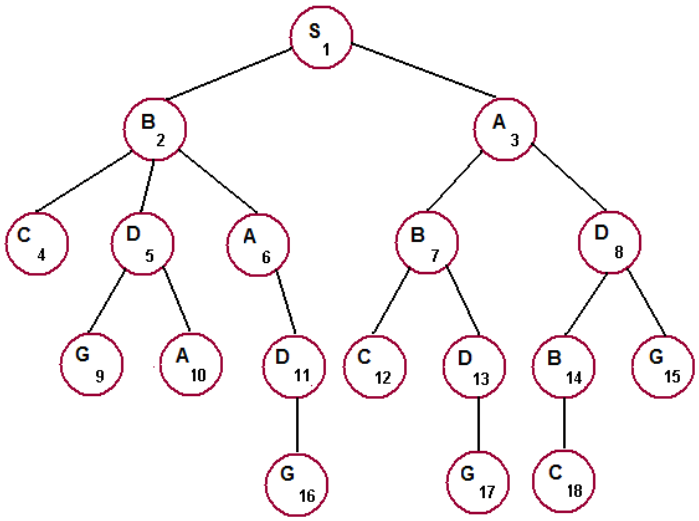

Figura 3.6. Recorrido de un grafo por el método simple Bradth first search.

- ✓ *Algoritmo de Dijkstra (Single-Source Shortest-Path)*: la función de costo es f(n) = g(n), donde g(n) = costo del camino entre el nodo inicio y el nodo actual.

  El algoritmo clásico de Dijkstra encuentra los caminos mínimos entre un vértice y todos los demás vértices del grafo pesado sin aristas de costo negativo, pero puede ser modificado para encontrar el camino de costo

mínimo entre un nodo de inicio y un nodo particular deteniendo el algoritmo una vez alcanzado dicho nodo. (Dijkstra, 1959).

- ✓ *A* (Best- First Branch and Bound*): la función de costo que utiliza suma el costo del camino entre el nodo inicial y el nodo actual, a la distancia estimada desde el nodo inicial al nodo solución. Garantiza encontrar la solución de menor costo.

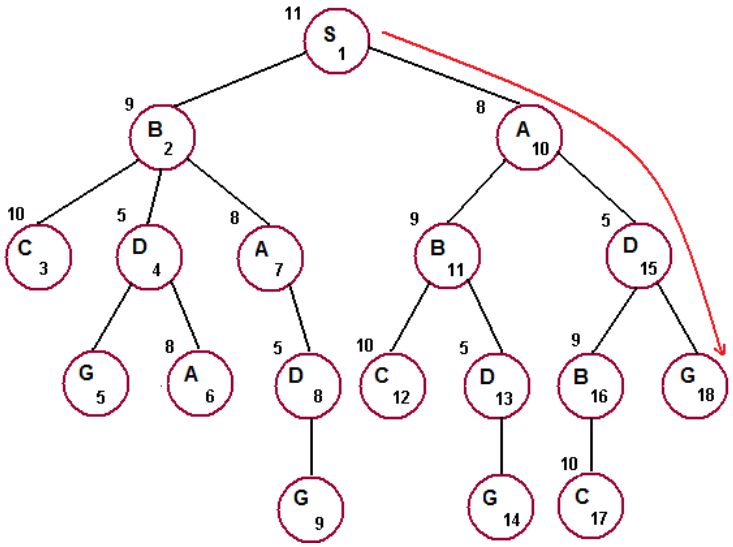

Figura 3.7: Recorrido de un grafo por el método Best- First Simple

En algunos problemas, un nodo puede ser alcanzado desde distintos caminos. Dado que los algoritmos DFS no soportan chequeos de ciclos, son en general aplicables a árboles. En el caso de aplicarlos a grafos cíclicos un mismo nodo podría ser procesado múltiples veces, haciendo que el grafo se aplane en un árbol, como se ha visto en los ejemplos.

La técnica BFS tiene como desventaja su uso exhaustivo de memoria, ya que debe almacenar la frontera de búsqueda en la lista abierta, requiriendo espacio de memoria frecuentemente exponencial.

En la figura 3.8 se muestra la clasificación de los algoritmos y se indica el tipo de solución encontrada.

Cabe destacar que de los algoritmos citados solo Ordered Backtracking, IDA*, DFBB, y A* utilizan una función heurística para tomar conocimiento de cuán cercano esta el nodo a procesar del nodo solución.

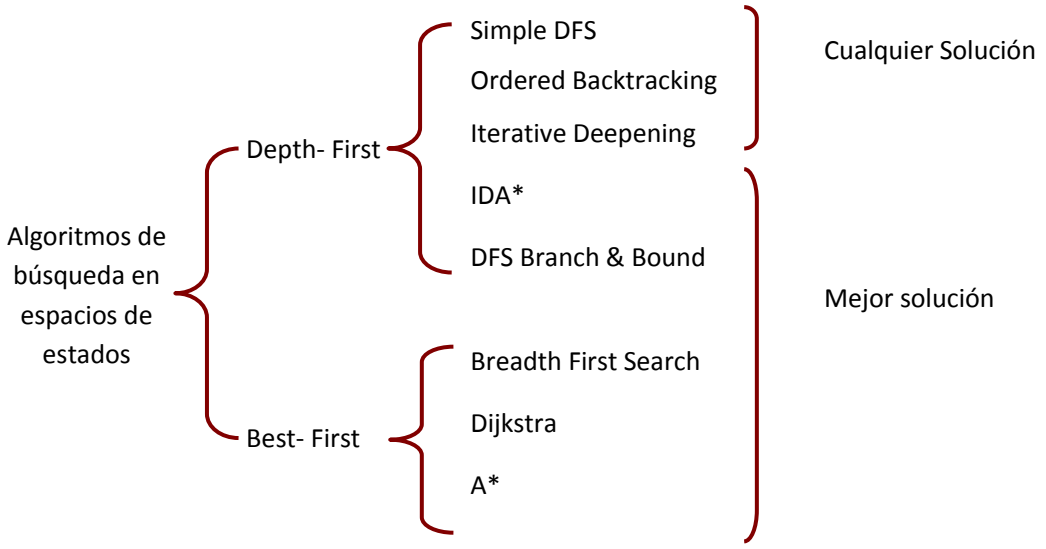

Figura 3.8: Clasificación de algoritmos de búsqueda en espacios de estados.

3.3 Análisis de los algoritmos vistos y elección

El algoritmo A* fue seleccionado como base para resolver el problema elegido como caso de estudio , ya que está garantizado que siempre encontrará la mejor solución, siempre y cuando la heurística a utilizar sea admisible.

Por otra parte, A* es el mejor algoritmo con dicha característica, ya que IDA* en cada iteración vuelve a realizar trabajo de la iteración anterior. Sin embargo, debido al uso eficiente que hace de la memoria, es utilizado frecuentemente para resolver instancias difíciles de problemas NP-Complejo, debido a que en algunos casos una búsqueda BFS agota la memoria disponible en cuestión de minutos.

Han sido descartados de antemano los algoritmos Simple DFS, Ordered Backtracking e Iterative Deepening, ya que no encuentran la solución requerida por el caso de estudio y además son aplicables a árboles. La técnica DFS Branch & Bound, además de tener este último inconveniente, debería procesar el espacio de búsqueda completo, siendo inaceptable el rendimiento que se obtendría.

La técnica Breadth First Search podría aplicarse para la resolución del problema del Puzzle. Al procesar el espacio de estados por niveles, si se encuentra una solución en el nivel actual esta sería la que requirió menor número de movimientos para resolver el tablero inicial. Sin embargo, esta variante puede tomar un tiempo considerable para alcanzar una solución, ya que suelen residir en zonas alejadas de la raíz del grafo, y para llegar a procesar dicho nivel debe procesar todos los nodos de los niveles anteriores.

El algoritmo de Dijkstra es un caso especial del algoritmo A*, que se obtiene al hacer h(n) = 0. La complejidad de A* es la misma que Dijkstra, pero es mejor en tiempo de ejecución para el caso promedio. En el caso del Puzzle, cada movimiento del hueco realizado representa una arista en el espacio de estados. Por lo tanto, el grafo implícito

tendrá aristas con costo 1, descartando por completo la aplicación del algoritmo de Dikjstra ya que estaría simulando un Breadth First Search, consumiendo un tiempo mayor.

Supongamos otro problema cuyo grafo implícito es pesado, y que la heurística utilizada por el algoritmo A* es exacta (caso ideal). En este caso todos los nodos en el camino hacia el nodo solución tendrían igual costo, lo que conduciría al algoritmo a procesar en cada iteración un nodo que está en el camino de la solución óptima. Dado que el algoritmo de Dijkstra visita los nodos en orden creciente de distancia desde el vértice inicial, el algoritmo perdería tiempo explorando otros nodos en dirección opuesta al nodo solución.

4. Funciones Heurísticas

Algunos de los algoritmos de búsqueda en árboles y grafos, y en particular, aquel tomado como base para resolver el problema del caso de estudio, utilizan información sobre el problema para guiar la búsqueda. Estas búsquedas son más eficientes que la búsqueda a ciegas, donde solo se generan nuevos estados y se verifican para detectar la solución.

Los algoritmos de búsqueda heurísticos valúan los nodos durante la búsqueda a partir de aplicarle una función heurística, y así procesan primero el nodo que aparenta ser el más prometedor. Dependiendo de la función utilizada, surgen las diversas variantes de la técnica Best-First.

En esta sección se estudian las funciones heurísticas, sus propiedades y como se obtienen a partir de una metodología general. Para finalizar, al final de la sección se estudian las funciones heurísticas basadas en base de datos.

4.1 Definición

Una función heurística h(n) es aquella comúnmente usada para agregar conocimiento del problema al algoritmo de búsqueda, y estima el costo de alcanzar el nodo solución desde un nodo n. Esta función deberá cumplir las siguientes restricciones:

- ✓ Si n es el nodo solución, luego h(n) = 0.
- ✓ Para todo nodo n que no sea el nodo solución, h(n) > 0.

Un nodo puede tener información adicional, además de la configuración del problema en particular (a veces denominada *estado*), por ello se debe destacar que la heurística sólo depende del estado del nodo, y no usa otra información.

4.2 Propiedades

El algoritmo A*, para detectar soluciones óptimas, exige una heurística *admisible*. Una heurística admisible es optimista, ya que siempre indica que el costo de resolver un problema es menor o igual (en caso de ser exacta) al costo real para resolverlo. Entonces, una heurística admisible nunca sobreestima el costo de alcanzar el nodo solución desde un nodo n cualquiera.

Otra propiedad exigida para búsquedas en grafos es la *consistencia* o *monotonía* de la función heurística. Una heurística es consistente si para cada nodo n, y sus sucesores n', se cumple $h(n) <= c(n,n',a) + h(n')$, donde c(n,n',a) indica el costo de pasar del nodo n a n' mediante la acción "a". Esto evita que un camino óptimo a un estado ya procesado sea descartado, si es que no fue generado antes que el camino no óptimo hacia el mismo estado.

4.3 Metodologías para generar funciones heurísticas

Dado un problema cualquiera, es interesante encontrar una heurística de forma sistemática, que satisfaga las propiedades de admisibilidad y consistencia.

Una metodología para generar nuevas heurísticas surge a partir de generar un problema simplificado, el cual se puede obtener mediante la eliminación de alguna o varias restricciones del problema original. La heurística será el costo de la solución óptima para el problema simplificado. [PEA94]

Una solución al problema original es también una solución al problema simplificado, y es al menos tan cara como la solución en el problema simplificado, por lo tanto la heurística que surge a partir de esta metodología es admisible. También puede demostrarse que es consistente [RUS03]

Para un problema en particular, pueden existir distintas funciones heurísticas, algunas más precisas que otras.

Una de las restricciones para el problema del Puzzle es que una ficha solo puede moverse a la posición ocupada por el hueco. Si se elimina esta restricción, una ficha se podría mover a cualquier posición adyacente. Una heurística exacta para resolver el problema simplificado es la suma de las Distancias de Manhattan para todas las fichas (SDM). Por lo tanto, SDM es una heurística admisible y consistente para el problema original del Puzzle.

Otra restricción que se puede descartar es la regla de desplazamiento horizontal o vertical. En este problema simplificado, una ficha podría tomarse desde su posición actual y directamente ser posicionada en su posición final. La heurística que surge es conocida como *Misplaced Tiles*, y se basa en contar el número de fichas que no están en su posición final.

El problema de la técnica anterior es que requiere mucho razonamiento acerca del problema específico. Una teoría más general propuesta por [KOR96], permite derivar la SDM a partir de búsquedas. Realizando una búsqueda para cada ficha, sin tener en cuenta las demás, empezando de su posición final y registrando la cantidad de movimientos a toda posición del tablero, se obtiene una tabla donde para cada posición de cada ficha nos indica su Distancia de Manhattan (*DM*) a su posición final. Sumando la DM de cada ficha del tablero, se obtiene la heurística SDM. En la práctica, no es necesario precalcular dicha tabla de distancias, ya que puede obtenerse mediante un simple cálculo.

Sin embargo, dicha heurística no es exacta, porque ignora las interacciones entre fichas. Se puede generalizar lo realizado anteriormente, pero repitiendo el proceso con todo posible par de fichas. En otras palabras, para cada par de fichas, y cada posible combinación de posiciones que puedan ocupar, se realiza una búsqueda desde sus posiciones finales y se cuenta el número de movimientos para alcanzar dicha configuración. El objetivo es encontrar el camino más corto desde el estado final a todas

las posibles posiciones de las dos fichas, solo teniendo en cuenta la interacción entre ambas. A este valor obtenido se lo conoce como *Pairwise Distance (PD)*.

Para la mayoría de los pares de fichas y sus posiciones, *PD* igualará la SDM de ambas fichas. Sin embargo, se encuentran tres casos donde *PD* excede la SDM. Estas tres mejoras se verán en detalle en la sección 6.

Así, con poco razonamiento acerca del dominio del problema, se pueden encontrar mejoras a una heurística a partir de búsquedas.

4.4 Heurísticas basadas en bases de datos de patrones

Una base de datos de patrones almacena estimaciones de los costos para resolver subproblemas específicos en un patrón de la solución dada. Esta idea se basa en que un problema puede ser descompuesto en subproblemas. A partir de lo visto en la sección 4.3, se sabe que el costo de una solución a un subproblema es una cota inferior del costo de la solución al problema completo.

Para generar la base de datos, primero se debe seleccionar un subconjunto de las propiedades de un estado final, y luego de enumerar todas las posibles combinaciones para ese subconjunto, se calculan las soluciones óptimas para dicho problema simplificado. [CUL98]

Por ejemplo: para el problema del Puzzle 15 se pueden tomar un subconjunto de fichas del estado final (1, 2, 3, 4, 5, 9, 13, y el hueco), y para toda combinación de las ubicaciones posibles de las fichas (subestado) se calcula el costo óptimo de alcanzar el patrón solución. La figura 4.1. (a) muestra un subestado de este problema simplificado y (b) el patrón solución.

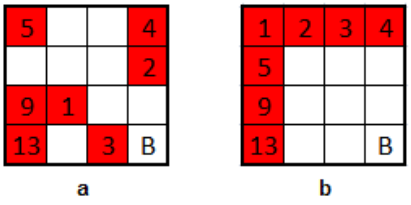

Figura 4.1: Patrón de un Puzzle 15
a. Subestado del problema simplificado **b.** Patrón de la solución

Durante la búsqueda, para valuar un nodo solo se tendrá en cuenta las posiciones de las fichas que son parte del subconjunto del patrón, de modo de realizar una búsqueda en la base de datos y encontrar la estimación de costo requerida.

A medida que se incluyen más fichas en el subconjunto que forma el patrón, más afinada será la heurística. Sin embargo, el cómputo de la base de datos tomará mayor cantidad de tiempo y su almacenamiento requerirá mayor espacio.

Otra estrategia es aprovechar la ventaja de dos o más bases de datos de patrones. Si las fichas de ambos patrones no son disjuntas, resolver el posicionamiento de las fichas en un patrón podría llevar a resolverlos en otro, por lo tanto en ciertos casos se podría llegar a sobrevaluar el nodo en caso de realizar una suma de las estimaciones debido a que se pueden repetir movimientos. La heurística debería tomar el costo máximo para el nodo a valuar en cuestión.

Otros autores [KOR00] proponen bases de datos de patrones disjuntos, que se basan en dividir el problema en regiones disjuntas y resolver en forma óptima solo el subproblema de ordenamiento de las fichas en cada región, contando los movimientos que son realizados por las fichas en esa región. Por ejemplo: en la heurística SDM, cada ficha es un patrón, y dado que son disjuntos se pueden sumar los costos para formar una heurística.

En la figura 4.2 se muestran ejemplos de regiones disjuntas en un tablero, que pueden usarse como patrones.

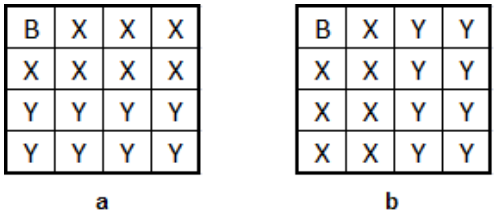

Figura 4.2: Cada tablero muestra ejemplos de patrones disjuntos

5. Arquitecturas Paralelas

Un usuario que se plantea desarrollar una aplicación paralela espera tener ciertas mejoras en el rendimiento con respecto a la aplicación secuencial respectiva. Dichas mejoras significarán, por ejemplo, reducir el tiempo de ejecución, resolver problemas más grandes en igual cantidad de tiempo, resolver un conjunto de instancias del problema en igual cantidad de tiempo, entre otras.

Un aspecto importante a tener en cuenta durante el desarrollo de la aplicación paralela es la arquitectura del sistema. El diseño de una aplicación paralela depende fuertemente de la arquitectura subyacente sobre la cual se quiera ejecutar la aplicación y del modelo de comunicación que se quiera utilizar para la interacción de las tareas, es decir memoria compartida o pasaje de mensajes. Existen también plataformas de procesamiento paralelo que permiten ambos mecanismos.

En los últimos años se ha generalizado el uso de arquitecturas distribuidas para el desarrollo de aplicaciones paralelas. La tendencia hacia el uso de arquitecturas tales como clusters, multi-clusters y sistemas GRID, cuya comunicación se basa en el modelo de pasaje de mensajes y cuyas redes de interconexión poseen diferentes topologías y características, es cada vez más común.

En esta sección se estudian las arquitecturas paralelas distribuidas nombradas anteriormente (clusters, multiclusters y sistemas GRID) y se plantean algunas líneas a tener en cuenta para mejorar el rendimiento de las aplicaciones paralelas que se ejecutan sobre las mismas.

5.1 Arquitectura tipo Cluster y Multi-Clusters

La arquitectura tipo cluster ha surgido como alternativa a los sistemas multiprocesadores de memoria compartida, y se han vuelto plataformas comunes en el cómputo paralelo de problemas complejos debido a sus ventajas en cuanto a su relación costo/rendimiento. La tendencia ha evolucionado de tal forma que las aplicaciones de cómputo intensivo aprovechan las ventajas de procesamiento creciente de las computadoras estándares de escritorio con costo aproximadamente constante y también la forma relativamente sencilla en las mismas pueden ser utilizadas para cómputo paralelo. [BAK99] [BAS99] [BOH02]

Un *cluster* es un sistema paralelo o distribuido formado por un conjunto de computadoras conectadas a través de una red, que se comportan como si fuesen un único recurso de cómputo. Su objetivo principal consiste en mejorar el rendimiento y/o la disponibilidad de un sistema, siendo más económico que una computadora de rapidez y disponibilidad comparables. [PFI98]

Los clusters pueden ser utilizados tanto para cómputo de problemas complejos en paralelo, haciendo que sus nodos trabajen en conjunto para resolver un problema que ha sido dividido en varios subproblemas pequeños; servidores web, diseñando el cluster para brindar alta disponibilidad, detectando y solucionando automáticamente fallos que puedan surgir; bases de datos de alto rendimiento, armando un cluster de servidores que trabajen al unisono, balanceando la carga y proveyendo alta disponibilidad; entre otras.

Entre los beneficios de la tecnología cluster se encuentran:

- ✓ Incremento de velocidad de procesamiento ofrecido por los clusters de alto rendimiento.
- ✓ Incremento del número de transacciones o velocidad de respuesta ofrecido por los clusters de balanceo de carga.
- ✓ Incremento de la confiabilidad y la robustez ofrecido por los clusters de alta disponibilidad

Las máquinas que conforman el cluster pueden ser homogéneas o heterogéneas. Este será un factor importante para el análisis del rendimiento que puede obtenerse.

Por otro lado, también es posible la ampliación de las ideas de cómputo paralelo a más de un cluster, dando origen al cómputo paralelo *intercluster*.

Un *multi-cluster* es un sistema distribuido que se obtiene al conectar dos o más clusters a través de redes locales (LAN) o extendidas (WAN). Un multicluster puede ser visto como un *cluster* con alto nivel de heterogeneidad, cuya potencia es la suma de las potencias de los clusters. Se debe lograr que el multicluster sea visto como una máquina virtual para la aplicación.

La heterogeneidad de un multi-cluster dependerá de los siguientes factores:

- ✓ Los clusters que lo conforman pueden ser heterogéneos, si las computadoras que lo integran no son idénticas.
- ✓ El sistema operativo de los componentes que lo integran puede ser común o no.
- ✓ Las redes de interconexión intra-cluster pueden tener distinto ancho de banda.
- ✓ Las redes de interconexión inter-cluster pueden tener distinto ancho de banda, y además podrían tener ancho de banda variable (típico de redes WAN sobre Internet).
- ✓ Todos los clusters se dedican a una la aplicación definida o lo comparten con otras tareas.

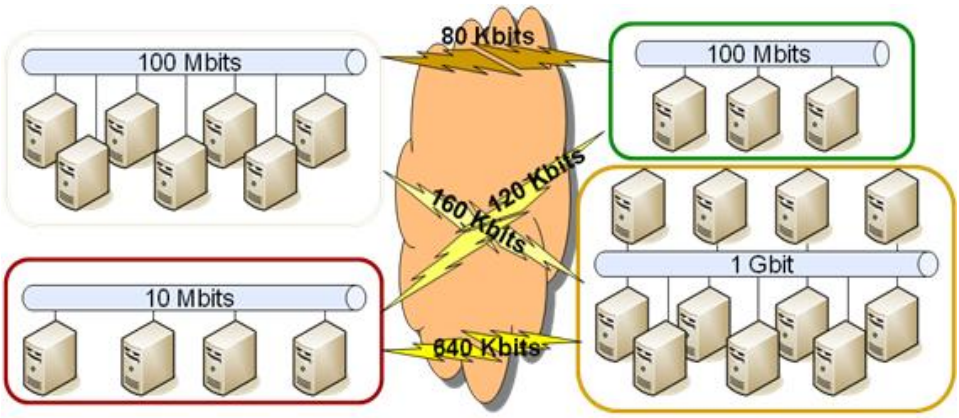

Figura 5.1. Multicluster heterogéneo

La figura 5.1. muestra un multicluster formado por 4 clusters, cada uno conecta componentes (homogéneos o heterogéneos) mediante una red local. Las redes locales de los diferentes clusters tienen distinto ancho de banda, e incluso las interconexiones entre clusters difieren en el mismo parámetro. De este modo, la heterogeneidad puede aparecer en distintos niveles: heterogeneidad en un cluster, heterogeneidad entre clusters, heterogeneidad en la comunicación entre clusters, heterogeneidad entre red local y red remota.

5.2 Bibliotecas y Lenguajes para el desarrollo de aplicaciones paralelas sobre un cluster

El modelo de pasaje de mensajes es un modelo de programación que consiste en procesos, que pueden ejecutarse o no en distintos procesadores, que se comunican intercambiando mensajes. Este modelo se aplica a máquinas paralelas de memoria distribuida, donde los procesadores no comparten memoria, ni existen mecanismos para leer la memoria del otro. [5]

Anteriormente, cada proveedor de hardware desarrollaba sus propias bibliotecas, con rendimiento óptimo para su propio hardware pero incompatible con las computadoras paralelas de otros proveedores.

El uso generalizado de los clusters como plataforma de cómputo paralelo ha generado el surgimiento de diversas especificaciones para bibliotecas de pasaje de mensajes explícito, que posibilitan implementar algoritmos en una forma portable. Estas bibliotecas son utilizadas en conjunto con lenguajes de programación estándar tales como C, C++, Fortran, Java, entre otros.

PVM y MPI son dos ejemplos de bibliotecas estándares para pasaje de mensajes.

[5] El modelo de pasaje de mensajes también puede ser implementado en máquinas de memoria compartida.

5.2.1 PVM

PVM fue diseñado para permitir que una red de computadoras heterogénea comparta sus recursos de cómputo (como el procesador y la memoria RAM) y sea usada como una única computadora paralela, con el fin de disminuir el tiempo de ejecución de un programa al distribuir la carga de trabajo en varias computadoras.

Algunas posibilidades de PVM son:

- ✓ Seleccionar un conjunto de maquinas y ser usado para ejecutar tareas, pudiendo añadir o eliminar maquinas dinámicamente.
- ✓ Cooperación mediante intercambio de mensajes explícito.
- ✓ Intercambiar mensajes conteniendo datos con tipos distintos, y entre maquinas con distintas representaciones de datos.
- ✓ Conectar máquinas heterógeneas, comunicadas mediante redes heterógeneas.
- ✓ Una tarea puede registrarse para ser notificada de eventos que cambien la maquina virtual.

La librería funciona en lenguajes como C, C++ (CPPVM) y Fortran, aunque existe una versión para JAVA llamada jPVM.

5.2.2 MPI

MPI (Message Passing Interface) es una biblioteca estándar, que dispone de diversas implementaciones para distintas máquinas, de modo que internamente aprovechan las características físicas del sistema. Por esta razón, MPI es generalmente más rápido que PVM.

MPI provee los siguientes grupos de funciones:

- ✓ Funciones para comunicación punto a punto.
- ✓ Funciones para comunicación entre grupos de procesos (comunicación colectiva).
- ✓ Capacidad para especificar topologías de comunicación lógica, que deben ser mapeados en la topología física de interconexión, resultando en un incremento de velocidad en la transferencia de mensajes.
- ✓ Capacidad para crear tipos derivados.

Existen distintas distribuciones de MPI, como por ejemplo LAMMPI, MPICH2 y JMPI (implementación de la librería para JAVA). Todas respetan el estándar, y en consecuencia el mismo código puede ser compilado en cualquiera de estas distribuciones (obviamente utilizando el mismo lenguaje de programación). Se debe

tener en cuenta que un programa compilado en una distribución no puede ser ejecutado en otra.

5.3 Sistemas GRID

Los clusters proveen alta potencia de cómputo para resolver problemas complejos de forma de obtener alto rendimiento con bajos costos. No obstante, algunas aplicaciones demandan potencia de cómputo, espacio y gestión de almacenamiento mayor a la que proveen los clusters, ya que gestionan grandes cantidades de datos y han de hacerlo de forma eficiente y sencilla.

A partir de las arquitecturas de cluster y multicluster surgen nuevas infraestructuras de cómputo ditribuido/paralelo.

Un GRID es un tipo de sistema paralelo distribuido que permite compartir, seleccionar y agregar recursos que están geográficamente distribuidos, tales como computadoras (ya sean PCs, clusters, multiclusters, supercomputadoras, etc), software, información, bases de datos, instrumentos (telescopios, radios, etc), dispositivos y personas. De este modo, la cantidad de nodos a conectar en un GRID es ilimitada, pudiendo estos ser heterogéneos, por lo que provee gran potencia, de forma flexible y escalable. En este caso, se posee bajo grado de acoplamiento de los procesadores y en general un bajo rendimiento de la red de interconexión.

Algunos de los objetivos de la infraestructura GRID son:

- ✓ Proporcionar acceso transparente a potencia de cálculo y capacidad de almacenamiento distribuida, posibilitando el funcionamiento de aplicaciones a gran escala.
- ✓ Facilitar el acceso de recursos compartidos desde un punto, de modo que el usuario tenga la ilusión de estar accediendo a un único recurso, y no a una infraestructura distribuida.
- ✓ Posibilitar compartir información y datos entre usuarios con intereses distintos, de modo que debe reunir centros de supercomputación diversos.

A diferencia de los clusters, un GRID puede realizar otras tareas independientes del algoritmo, brindando alta disponibilidad de procesamiento y/o almacenamiento.

Hoy en día, la principal aplicación de la tecnología Grid es la supercomputación que se enfoca en resolver problemas tipo “gran desafío” (grand challenge) como pueden ser: la simulación del clima a escala planetaria, análisis del genoma, simulación de procesos biológicos del ser humano, estudio de modelos de catástrofes naturales, análisis de datos epidemiológicos, estudios de astronomía, etc. [JOS03] [JUH04] [OGU03]

5.4 Migración de aplicaciones desde un cluster a multi-cluster

La migración de una aplicación de un cluster a un multi-cluster debe perseguir el objetivo de reducir el tiempo de ejecución, sin dejar de lado la utilización eficiente de los recursos, es decir se debe garantizar un umbral de eficiencia de utilización de los recursos. Por ello, es probable que no se tenga que usar todos los nodos del multi-cluster, sino se deberá seleccionar un subconjunto del mismo de tal forma que se logre la eficiencia deseada y se reduzcan los tiempos.

Además se debería evitar la introducción de grandes cambios en la aplicación, permitiendo que la misma utilice cualquier configuración de cluster/multi-clusters, no limitando la cantidad de recursos que se podrían utilizar.

Por otra parte, se debería garantizar robustez en la aplicación, superando los problemas de interconexión que aparecen al usar una red no dedicada como Internet. Por ello, en cada cluster, se podría dedicar un nodo a un proceso gestor de comunicaciones, que se encargue de realizar las comunicaciones entre clusters (por ejemplo: recibir y enviar trabajo para balancear la carga), de forma de proveer transparencia.

Realizando un análisis de la aplicación en cuestión, basándose en su esquema de resolución, se podría hacer un modelo para estimar el tiempo de ejecución que tenga en cuenta el tiempo de cómputo y tiempo de comunicaciones intra-cluster e inter-cluster, para lo que se deberá calcular la potencia del multicluster en cuestión y la capacidad de comunicaciones del sistema.

Con esta metodología, se podría predecir el tiempo de ejecución en un multicluster y y la eficiencia a obtener, así se podrá realizar una selección de los recursos para lograr la eficiencia requerida y al mismo tiempo mejorar el tiempo de ejecución.

Durante el análisis de la aplicación se deberían encontrar los cuellos de botella. Las comunicaciones inter-cluster son más costosas, por lo que se deberían reducir. Otro parámetro importante a analizar será el balance de carga entre los clusters.

6. Desarrollo

En esta sección, se presenta el algoritmo secuencial para resolver el problema del Puzzle N^2-1, junto con las funciones heurísticas utilizadas en la búsqueda.

Se expone además el algoritmo paralelo desarrollado para una arquitectura tipo cluster, y se discuten las técnicas de balance de carga y detección de terminación utilizadas.

Al final de la sección se plantean decisiones tomadas para la implementación de ambos algoritmos.

6.1 Funciones heurísticas para el problema del Puzzle

Las funciones heurísticas se utilizan durante la búsqueda para estimar el costo de un nodo, es decir indicará cuán cercano se encuentra al nodo solución. A medida que la heurística se vuelve más afinada, la estimación del costo de los nodos es más próxima al costo real, por lo que los algoritmos que la utilizan tenderán a procesar menor cantidad de nodos.

Las heurísticas son dependientes del dominio del problema, por lo que ha sido de gran interés descubrir métodos generales para generar heurísticas para un problema. En especial, para el problema del Puzzle se han desarrollado diferentes heurísticas admisibles[6] que mejoran a la heurística clásica [KOR96]

Para el caso de estudio, la estimación heurística para un tablero indicará el número mínimo de movimientos requeridos para transformar dicho tablero en el tablero solución.

6.1.1 Suma de las Distancias de Manhattan (SDM)

La heurística clásica utilizada para el problema del Puzzle N^2-1 es la Suma de las Distancias de Manhattan *(SDM)* de todas las fichas del tablero a estimar *x* respecto al tablero final *s*.

La Distancia de Manhattan (*DM)* para una ficha representa la mínima cantidad de movimientos que se debe hacer para trasladar la ficha desde su posición actual a la posición en la que debe aparecer en el tablero final, y se calcula de la siguiente manera:

> *Supongamos que cada posición del tablero se representa como un par ordenado. Sea una ficha cualquiera r del tablero x, si la misma se encuentra en la posición (i,j) y en el tablero final s debe localizarse en el casillero (k,l), entonces DM(r)= $|i-k|+|j-l|$.*

[6] Una heurística admisible es aquella que nunca sobreestima el costo del nodo a valuar. Sea x un nodo y sea u el costo real para alcanzar s desde x, entonces h (heurística) es admisible si h(x) <= u, para todo x.

La figura 6.1 muestra un tablero donde N=4, y se indica la SDM para el mismo respecto al tablero final clásico. Las distancias de Manhattan de cada ficha son DM(1,5,10,12,15)=0, DM(4,6,7)=1, DM(2,11,14)=2, DM(13)=3, DM(3,9)=4 y DM(8)=5, por lo que la SDM(t)=25.

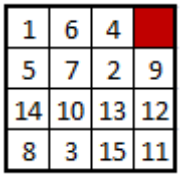

Figura 6.1: Tablero con N=4 y SDM=25.

La SDM es una heurística admisible, ya que asume que las fichas pueden moverse independientemente, pasando incluso por encima de otras, quitando la restricción de movimiento legal planteado en 2.2.2.

6.1.2 Detección de Conflictos Lineales

Un *"conflicto lineal"* entre dos fichas *x* e *y* ocurre cuando las mismas están posicionadas en su fila correcta pero invertidas. En este caso, una de ellas deberá cambiar de fila para que la otra pueda trasladarse hasta su posición final y, luego de desplazarse a través de las columnas tal como lo indica su DM, la ficha debe retornar a su fila destino. Lo mismo puede aplicarse a las columnas. Así, por cada conflicto lineal, se podría agregar 2 movimientos adicionales a la SDM del tablero.

Una pieza podrá formar parte de a lo sumo un conflicto lineal por fila y uno por columna. Esta restricción evita que la heurística sobreestime el costo real, dejando así de ser admisible.

La Figura 6.2 presenta un tablero con N=4 y un conflicto lineal entre las fichas 1 y 2 (respecto al tablero final clásico). En este caso, DM(1)=2 y DM(2)=1. La SDM parcial para estas fichas sería 3, costo no real ya que la ficha 1 debería pasar por encima de la ficha 2, y viceversa. Se muestra además la secuencia de movimientos que se debería realizar ante el conflicto lineal.

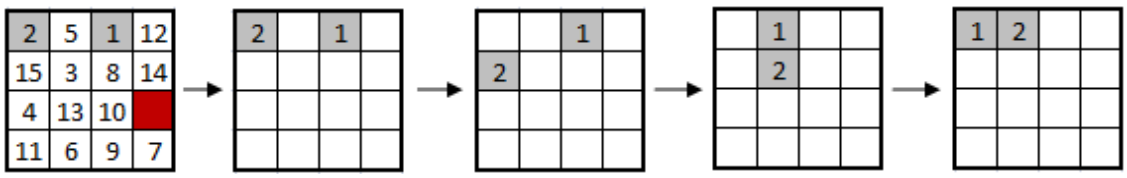

Figura 6.2: Secuencia de movimientos ante un conflicto lineal en una fila.

6.1.3 Suma de las Distancias de Manhattan y Detección de Conflictos Lineales (SDMCL)

La heurística de predicción de trabajo analizada en este trabajo (SDMCL) se basa en lo propuesto por Hanson [HAN92] y combina la distancia de Manhattan y la detección de conflictos lineales, definiéndose con la fórmula 1.

$$SDMCL\ (t) = SDM(t) + CL(t) \qquad (1)$$

Donde t representa el tablero actual, SDM es la función que calcula la suma de las Distancias de Manhattan para t, y CL(t) = (cantidad de conflictos lineales en t) * 2.

Claramente, la suma de cantidades por conflictos lineales preserva la admisibilidad de la heurística.

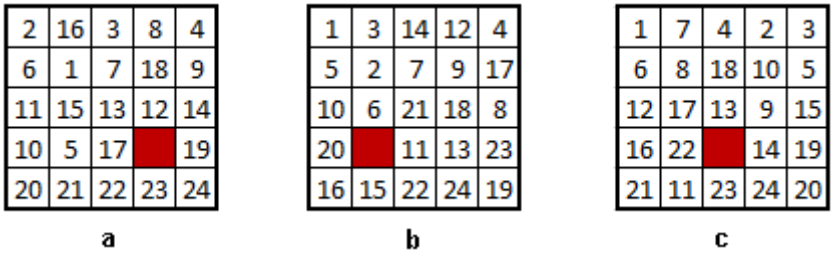

Figura 6.3. Tableros con N=5, cuya SDMCL respecto a distintos tableros finales es mayor que la SDM.

En la figura 6.3 se visualizan tres tableros con N=5. Para el tablero (a) la SDM=20 y la SDMCL=24 (respecto al tablero final mostrado en la figura 6.4.a), el tablero (b) tiene SDM=34 y SDMCL=36 (respecto al tablero final que se visualiza en 6.4.b), y para el tablero (c) la SDM=25 y la SDMCL=29 (respecto al tablero final que se muestra en 6.4.c).

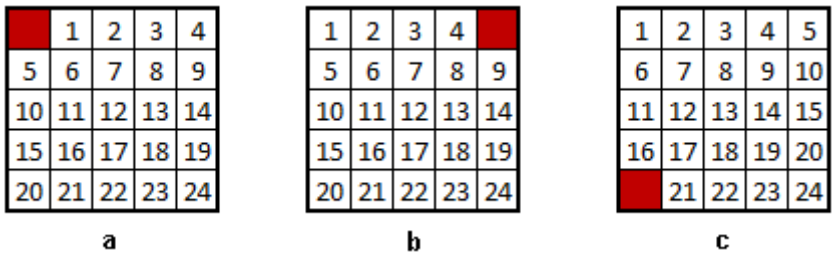

Figura 6.4. Tableros finales para los tableros iniciales de la figura 6.3.

6.1.4 Últimos Movimientos ("Last Moves")

El último movimiento que se realiza para resolver un puzzle siendo N=4 y con el tablero final mostrado en la figura 6.5.a es trasladar la ficha 1 desde la posición (1,1) a la posición (1,2) o mover la ficha 4 desde la posición (1,1) a la posición (2,1). De lo anterior se puede decir que antes del movimiento final la ficha 1 o la 4 deben estar en la esquina superior izquierda del tablero.

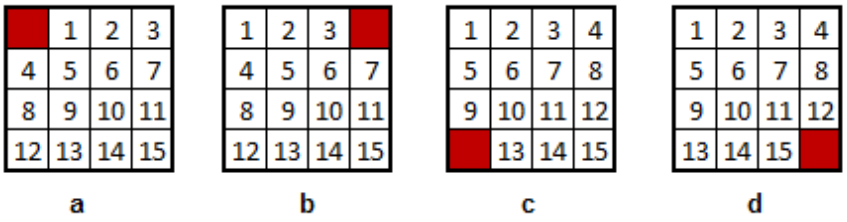

Figura 6.5: Ejemplos de tableros finales con posibilidad de aplicar la heurística "Últimos movimientos"

El mismo concepto puede ser aplicado con cualquier tablero final que tenga el hueco en alguna de las esquinas, y para cualquier tamaño del tablero. En particular, es interesante destacar que la heurística es aplicable a los tableros finales que se visualizan en la figura 6.5.b, 6.5.c, 6.5.d.

Dado que la DM para una ficha se calcula desde su posición actual a la posición final, si la ficha 1 no está en la primera columna o la ficha 4 no está en la primera fila, la SDM no es real ya que el camino para dejar alguna de estas fichas en su posición final no haría que la misma pase por la esquina superior izquierda. Por consiguiente, si la ficha 1 no se encuentra en la primera columna y la ficha 4 no se encuentra en la fila superior, se puede sumar dos movimientos adicionales a la SDM, preservando la admisibilidad de la heurística.

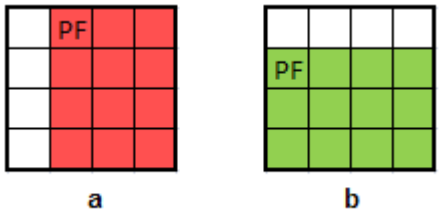

Figura 6.6: Tablero en el que la DM de las fichas 1 y 4 no acomoda el paso de la ficha por la esquina

La figura 6.6. muestra la zona en la que debería estar la ficha 1 (a) y la ficha 4 (b) junto con un indicador de su posición final (PF). La DM calcula la mínima cantidad de pasos para mover una ficha a su posición final, por lo que si la ficha 1 está en la zona indicada con rojo, su DM acomodará el paso de la ficha por esa zona. Algo parecido ocurre con la ficha 4. Al sumarse dos movimientos se haría pasar a alguna de estas fichas por la esquina superior izquierda.

En caso de combinar la *SDMCL* con la mejora planteada debe tenerse en cuenta ciertas interacciones entre ambas. Supongamos que la ficha 1 y la 4 no están en la primera columna y fila respectivamente. Si la ficha 1 está en su columna respectiva y se encuentra en un conflicto lineal con otra ficha, se suman dos movimientos por conflicto lineal. Pero esto podría provocar que la ficha 1 se mueva a la primera columna, y así podría pasar por la esquina superior izquierda del tablero. Un caso similar ocurre con la ficha 4. Por lo tanto, si la ficha 1 está en conflicto lineal en la columna o la ficha 4 está

en conflicto lineal en la fila entonces no se puede sumar dos movimientos adicionales por "últimos movimientos".

6.1.5 Fichas de las Esquinas del Puzzle ("Corner Tiles")

Esta mejora de la heurística se centra en las esquinas del tablero.

Por ejemplo, tomando en cuenta el tablero final de la figura 6.5.a, si la ficha 2 está en su posición correcta, y la posición de la ficha 3 está ocupada por otra ficha distinta, la ficha 2 tendrá que moverse temporalmente para dejar pasar la ficha 3 a su posición final. Esto requiere dos movimientos adicionales. Si se quiere combinar con las heurísticas anteriores, el 2 no debería estar en conflicto en la fila[7], ya que se habrían sumado dos movimientos. La misma regla se puede aplicar al 7, a menos que este en conflicto en la columna. Si ocurren ambos casos, es decir el 2 y el 7 están en su posición y el 3 no, entonces se podrían sumar a la estimación heurística base 4 movimientos.

Las mismas reglas se aplican a las demás esquinas, no siendo así para la esquina que contiene el hueco en el tablero final[8].

6.2 Solución secuencial

A* es una variante de la técnica de búsqueda Best First Search [REI93], en la cual cada nodo n se valúa de acuerdo con el costo de alcanzar el mismo a partir de la raíz del árbol de búsqueda ***(g(n))*** y una heurística que estima el costo para ir de n hasta un nodo solución ***(h(n))***. Luego la función de costo será ***f(n)*** = g(n) + h(n). Si la heurística es admisible[9], está garantizado que el algoritmo A* siempre encontrará la mejor solución.

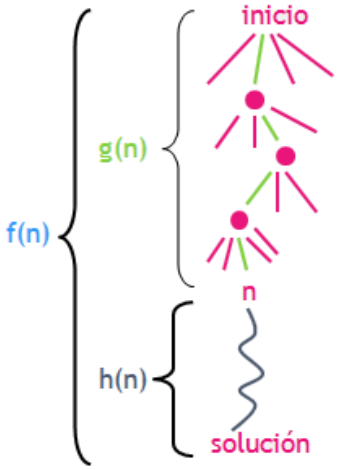

Figura 6.7: Función de estimación para los nodos en el algoritmo A*.

[7] Nunca se dará el caso en que la ficha 2 estando en su posición final esté en conflicto en la columna.
[8] Esto se debe a la difícil interacción de esta heurística con la de últimos movimientos.
[9] Una heurística admisible es aquella que nunca sobreestima el costo del nodo a valuar.

La figura 6.7. muestra la función de estimación f(n) usada para evaluar los nodos en el algoritmo A*.

- ✓ g(n) es la distancia (conocida) desde el nodo inicio al nodo n.
- ✓ h(n) es la función heurística que calcula la distancia (estimada) desde n al nodo solución.
- ✓ f(n) es una estimación de la distancia del camino desde inicio a solución, pasando por n.

El algoritmo desarrollado mantiene una lista de nodos no explorados (lista abierta[10]), ordenada de acuerdo al valor de la función f, y otra lista de nodos ya explorados (lista cerrada[11]), utilizada para evitar ciclos en el grafo de búsqueda. Inicialmente la lista abierta contiene un solo elemento, el nodo inicial, y la lista cerrada está vacía.

En cada paso, el nodo con menor valor f (el nodo más prometedor) es removido de la lista abierta y es examinado. Si dicho nodo es la solución, el algoritmo termina. En caso contrario, el nodo es expandido (generando los nodos hijos a partir de aplicar movimientos legales) e insertado en la lista cerrada. Cada nodo sucesor es insertado en la lista abierta si no ha sido procesado con anterioridad. En caso de haber sido procesado pero su valor de costo mejora al anterior, este nodo será aceptable.

```
Crear lista abierta.
Crear lista cerrada.
Insertar (lista abierta, ti, h(ti,tf)).
// h es la función heurística, que estima el costo de moverse desde el nodo x al nodo y.
mientras (lista abierta no vacía) y (no encontré solución)
    // Extraigo el mínimo nodo de la lista abierta, llamémoslo n
    n = EliminarMínimo (lista abierta)
    // Si h(n) = 0  termina el algoritmo, sino se expande n.
    si (EsSolución(n))
        solución = n
    sino
        hijos = Expandir(n)
        Insertar(lista cerrada,n)
// Un nodo es aceptable si no está en la lista cerrada, o si está  pero con un costo mayor al
actual para cada hijo de n
        Para cada nodo hijo en hijos
            si (EsAceptable(hijo))
                Insertar(lista abierta, hijo, h(hijo) + g(n) + 1)

Retornar solución.
```

Figura 6.8: Algoritmo secuencial.

[10] La lista abierta recibe su nombre por razones históricas, pero es implementada usando una minheap.
[11] La lista cerrada recibe su nombre por razones históricas, pero es implementada con una tabla de hash.

La figura 6.8. muestra el esquema de solución secuencial. La entrada de dicho algoritmo es ti (tablero inicial) y tf (tablero final). La función heurística (h) estima la distancia entre un nodo n y el tablero final tf. Antes de la ejecución de este algoritmo, se debe verificar si el tablero final tf es alcanzable desde el tablero inicial ti. Esto se realiza aplicando el algoritmo planteado en la sección 2.2.3.

Dado que siempre se remueve el nodo con menor valor f, la lista abierta comúnmente se implementa con una cola de prioridades.

La lista cerrada se puede implementar con una tabla de hash donde cada entrada contiene como clave un valor de la heurística h(x), que identifica un conjunto de tableros, y como valor una lista con todos los tableros cuyo h(x) es igual a la clave. Dado que un tablero siempre tendrá un único valor h(x), esta estructura permite hacer búsquedas rápidas. Un mismo tablero puede aparecer en distintos niveles del grafo, por lo tanto no sería útil poner como clave f(x).

Si se requiere conocer el camino desde el nodo inicial hasta el nodo solución se debe seleccionar alguna de las siguientes opciones para la implementación:

- ✓ Cada nodo mantiene una referencia a su nodo padre. Así para obtener la salida para el usuario se debería hacer un recorrido a partir del nodo solución y siguiendo punteros.
- ✓ Cada nodo posee la secuencia de pasos por la cual se llegó hasta él, así se podrá determinar cómo fue encontrado.

En el problema del Puzzle, el usuario quiere conocer la secuencia de movimientos del hueco que debe hacer para alcanzar el tablero final a partir del tablero inicial. Dado que el hueco se puede mover a sus posiciones adyacentes, la salida del algoritmo será una secuencia de tokens (arriba, abajo, izquierda, derecha), pudiendo implementar cualquiera de las soluciones anteriores debido a la escasa memoria que requieren.

En particular se optó por la segunda opción para independizar un nodo de los demás. Una optimización realizada es la liberación del espacio de memoria usado para mantener la secuencia de pasos de un nodo que será insertado en la lista cerrada.

Se comenta a continuación la funcionalidad de los distintos módulos del algoritmo:

- ✓ EsSolucion(n): chequea en tiempo constante si el nodo n es la solución. Esto se comprueba verificando si h(n) = 0.
- ✓ Expandir(n): aplica los movimientos legales posibles al nodo n, generando sus sucesores.
- ✓ EsAceptable(n): chequea si el nodo debe ser insertado en la lista abierta.
- ✓ $h(n_1,n_2)$: representa la función que calcula el costo estimado de ir desde el nodo n_1 al nodo n_2.

6.3 Algoritmo Paralelo

La paralelización natural del problema consiste en iniciar la evolución de diferentes nodos "posibles" sobre los distintos procesadores del cluster, de esta manera se distribuye la lista abierta y cerrada entre los procesadores. Esta solución propuesta es descentralizada, es decir cada proceso realizará una búsqueda en una parte del espacio de estados del problema no requiriendo un proceso central.

Por otra parte, el balance de carga es un aspecto muy importante en esta clase de problemas. Dado que el grafo es implícito y generado durante la ejecución del algoritmo, se debe adoptar una técnica dinámica de distribución de carga (ya sea centralizada o distribuida).

A medida que el algoritmo progresa es necesario comunicar los procesadores para informar resultados parciales alcanzados y posibilitar la detección de terminación de la búsqueda, o bien descartar soluciones, de acuerdo a la métrica elegida, que no mejorarán la solución parcial encontrada hasta el momento.

A continuación, se estudiarán las técnicas para lograr balance de carga, el algoritmo seleccionado para detectar terminación y el algoritmo paralelo que resuelve el problema del Puzzle.

6.3.1 Balance de carga

Las técnicas de balanceo de carga tienen como objetivo primordial igualar el uso de los procesadores, de forma de incrementar la eficiencia del sistema paralelo[12]. Estas técnicas se basan en distribuir las tareas entre un conjunto de procesadores, de forma que resulte en que cada uno tenga una cantidad de trabajo cuyo procesamiento requiera aproximadamente un tiempo de resolución semejante.

La performance y escalabilidad de estas estrategias dependerán de la arquitectura del sistema. Dependiendo del problema a tratar y de la arquitectura utilizada se usarán unas técnicas u otras para reducir el número de comunicaciones y balancear la carga, es decir, reducir el tiempo de ejecución del programa paralelo.

Las técnicas de balance de carga pueden clasificarse en dos grupos:

- ✓ *Estáticas:* las tareas se reparten antes de comenzar el cómputo, esto es aplicable cuando el tiempo de cómputo de las tareas es conocido a priori.
- ✓ *Dinámicas:* si la carga de trabajo varía durante la ejecución y no puede estimarse de antemano, se realizan etapas de balanceo durante la ejecución.

[12] Sistema paralelo es la combinación del algoritmo paralelo y la máquina sobre la que se ejecuta.

En general, las técnicas de balance de carga dinámicas son más complejas, pues deben tener en cuenta la capacidad de cómputo de los procesadores, más aún en un ambiente heterogéneo, e intentar reducir el tiempo de ociosidad del procesador, así también como la comunicación.

Estas técnicas se pueden dividir en dos grupos, dependiendo si la acción es iniciada por el receptor o por el emisor de la carga de trabajo.

En el primer caso, cuando un procesador se vuelve ocioso, es decir se queda sin trabajo para procesar, selecciona un procesador para que le done trabajo. Algunas de estas técnicas son: *Centralized Server, Global Round Robin, Asynchonous Round Robin, Random Polling* y *Nearest Neighbor*. Las primeras dos opciones son centralizadas y las técnicas restantes son distribuidas, es decir no requieren de un proceso central. En general las técnicas centralizadas se vuelven menos efectivas a medida que se incrementa el número de procesadores, ya que todos los procesadores interactúan contra un procesador, haciendo que este se vuelva un cuello de botella.

Las estrategias de balance de carga dinámicas distribuidas se basan en que el procesador ocioso seleccione, siguiendo una de las estrategias comentadas, un proceso *donador* de trabajo. Si dicho proceso dispone de trabajo, le envía parte de su carga al proceso que lo requiere. Caso contrario, le envía un mensaje de rechazo. En caso de recibir un mensaje de rechazo, el procesador ocioso busca otro *donador.*

En el segundo caso, un procesador genera trabajo y selecciona procesadores para enviarles parte de dicha carga. Algunas estrategias para seleccionar el procesador a quien se envía son *Near Neighbor*, *Random Sampling*, y *técnicas jerárquicas* (con superservidores, servidores y clientes).

En el caso de los algoritmos paralelos de búsqueda en espacios de estados es importante tener un buen balanceo del número de nodos a evaluar por cada procesador. En estos algoritmos, cuando se distribuye la carga, no solo hay que tener en cuenta la cantidad de nodos se envían o reciben, también se debe considerar la calidad de dichos nodos. En caso de pasar nodos que ya se sabe no conducen a una solución mejor que la solución parcial encontrada, el nodo receptor se quedará ocioso rápidamente, por lo que una estrategia debe adoptarse.

A continuación se explican las técnicas de balance de carga elegidas para la implementación del algoritmo.

6.3.1.1 Asynchronous Round Robin

Cada procesador mantiene una variable *target* que indica el proceso seleccionado como donador de trabajo. Dicha variable es inicializada en ((label + 1) modulo p), donde *label* es el identificador del proceso y p la cantidad de procesos. El valor de target es incrementado (modulo p) cada vez que se realiza un pedido.

Cuando un proceso se vuelve ocioso, envía un "mensaje de petición" de trabajo al proceso donador y espera una respuesta. Si el proceso recibe un mensaje de "rechazo", debido a que el proceso donador no tenía trabajo, envía una petición al próximo donador.

Un proceso que esta ocioso, puede ser elegido como donador por otros, por lo que al recibir "mensajes de petición" responderá con un "mensaje de rechazo".

6.3.1.2 Estrategia de división de trabajo

Cuando un proceso recibe un "mensaje de petición" y dispone de trabajo, se convertirá en donador del proceso que realizó el requerimiento. Si el donador le pasa poco trabajo o nodos que no conducirán a una solución mejor a la encontrada al momento, el receptor se quedará ocioso de nuevo rápidamente, por lo tanto la calidad de los nodos a enviar es importante para el balance de carga.

La estrategia adoptada es la siguiente: el donador enviará parte de sus mejores y peores nodos, cuyo costo sea inferior al costo de la mejor solución encontrada al momento.

6.3.2 Algoritmo de terminación modificado de Dijkstra

El algoritmo de terminación modificado de Dijkstra es utilizado para detectar terminación en un ambiente distribuido donde se está utilizando una técnica de balance de carga dinámica distribuida. [DIJ80]

Un proceso puede estar en uno de los siguientes estados en un momento dado: activo u ocioso. Sólo los procesos activos pueden enviar mensajes a los demás.[13] Luego de recibir un mensaje, si el proceso estaba ocioso se vuelve activo.

El estado del sistema en el cual todos los procesos están ociosos y no hay mensajes circulando por la red recibe el nombre de estado de *terminación*. El propósito del algoritmo es que un proceso detecte dicho estado.

Los procesos se consideran conectados en un anillo lógico y se pasarán un token.

6.3.2.1 Algoritmo Básico

Supongamos que un procesador que se vuelve ocioso nunca más recibirá trabajo. El procesador P_0 inicia la prueba de terminación cuando no tiene más trabajo, pasando el token a P_1. Cada procesador P_i que recibe el token lo mantiene hasta que se queda ocioso, pasándolo al procesador siguiente (en caso de ser P_N lo pasa a P_0). Cuando P_0 recibe el token nuevamente, quiere decir que todos terminaron su cómputo, por lo tanto detecta la terminación del algoritmo.

[13] Para la solución planteada, sólo se considerará como "mensaje" al mensaje de trabajo, ya que se pretende detectar el momento en que no hay mensajes de trabajo en circulación.

Si un procesador ocioso puede recibir trabajo de otro procesador predecesor en el anillo, es decir se admiten transferencias desde P_i a P_j si $i < j$, este algoritmo sigue siendo adecuado. Caso contrario debe aplicarse el algoritmo modificado explicado a continuación, ya que cuando P_j envía trabajo a P_i el token debe recorrer otra vez el anillo.

6.3.2.2 Algoritmo Modificado

El token de terminación consta de los campos "color" y "contador de mensajes"[14]. Cada proceso mantiene un contador c (inicialmente en cero). Al enviar un mensaje, se incrementará c en uno; al recibir un mensaje se decrementará c en uno. La suma de todos los contadores de los procesos es el número de mensajes pendientes en la red.

Cuando el proceso 0 se vuelve ocioso, envía el token con valor 0 en su contador de mensajes y color BLANCO. Cada proceso mantiene el token hasta que se vuelve ocioso, y luego lo pasa al proceso siguiente incrementando el contador de mensajes del token en c.

Cada proceso mantiene un color (inicialmente BLANCO). Cuando un proceso envía o recibe un mensaje, se torna NEGRO. Cuando un proceso pasa el token, se vuelve BLANCO. Si un proceso NEGRO pasa el token, luego el token se torna NEGRO, caso contrario mantiene el color.

Cuando el proceso 0 recibe el token nuevamente, lo examina y detecta terminación si:

- ✓ El proceso 0 está ocioso y su estado es BLANCO.
- ✓ El color del token es BLANCO.
- ✓ La suma entre el contador de mensajes del token y c es cero.

Caso contrario, el proceso 0 inicia una nueva vuelta de detección de terminación.

[14] El campo "contador de mensajes" se incluye para evitar problemas debido al orden en que se reciben los mensajes de trabajo y el token. Si un proceso envía un mensaje de trabajo a un proceso sucesor, y luego se queda inactivo, al recibir el token podría pasarlo hacia adelante. El proceso receptor podría recibir y pasar el token estando ocioso, y luego recibir trabajo, por lo que se podría detectar terminación aun habiendo trabajo para realizar.

6.3.3 Esquema de resolución del problema del Puzzle sobre un cluster

La estrategia de paralelización consiste en mantener las listas abierta y cerrada locales en cada procesador. Al principio sólo un procesador (Worker 0) tendrá como trabajo el nodo inicial, quien también se encargará de detectar la terminación de la búsqueda. A medida que son generados otros nodos, los procesadores recibirán los mismos para comenzar a trabajar.

Todos los procesadores harán una búsqueda en el ámbito local, construyendo su propia lista cerrada, para evitar trabajo repetido localmente, como también su lista abierta. Los procesadores deben comunicarse los valores mínimos de las soluciones encontradas, a fin de minimizar búsquedas innecesarias.

Para la implementación del algoritmo paralelo se utiliza la técnica de balance de carga dinámica distribuida "Asynchronous Round Robin" (ARR) y el algoritmo de "Terminación de Dijkstra" modificado para la detección de terminación.

Se utilizó un algoritmo de poda global, donde cada uno de los p procesos "workers" dispondrá un valor indicando el costo de la mejor solución encontrada hasta el momento (CMS), que se utilizará para acotar la búsqueda.

Un proceso que posee trabajo en su lista abierta a lo sumo procesa una cantidad fija de nodos en cada iteración (LW) o procesa nodos hasta encontrar una solución o que se vacíe su lista abierta. A continuación el "worker" recibe, si los hay, costos de "mejores soluciones" encontradas por los demás y a medida que esto ocurre actualiza (si es necesario) su variable CMS. De esta manera los nodos a procesar serán sólo los que tengan costo menor a CMS.

Si el proceso todavía posee trabajo en su lista abierta, verifica si ha recibido pedidos de trabajo de otros procesos, caso en el cual envía nodos del principio y final de su lista abierta al trabajador solicitante ocioso. Luego continúa trabajando sobre sus nodos.

En caso contrario, el proceso está ocioso, por lo que envía un pedido de trabajo a su donador siguiendo el algoritmo ARR. Si había encontrado una nueva solución, envía a todos los demás procesos el costo de la misma. Luego espera los tipos de mensajes enumerados a continuación, los cuales serán atendidos sin prioridad alguna:

- ✓ *Petición de trabajo:* un trabajador ocioso seleccionó a este proceso como su donador.
- ✓ *Trabajo:* el donador envía el trabajo requerido. Ahora el proceso está activo nuevamente.
- ✓ *Rechazo de pedido de trabajo:* el donador seleccionado no tiene trabajo. El proceso debe enviar un mensaje de pedido de trabajo al próximo donador.
- ✓ *Token:* recepción del token para detección de terminación. Si es necesario se actualiza el token y se pasa al siguiente proceso. El proceso 0, al recibir el

token, verifica si debe terminar la búsqueda y en ese caso envía un mensaje a los demás procesos avisando el fin del cómputo.

- ✓ *Nueva solución encontrada:* en caso de ser necesario, se actualiza la variable CMS.

Se utiliza el token de terminación para trasladar los movimientos de la solución de costo mínimo hasta el proceso 0, de forma que los mensajes de comunicación de nuevas soluciones encontradas durante el algoritmo sólo posean un valor entero, el costo, evitando así overhead de comunicación.

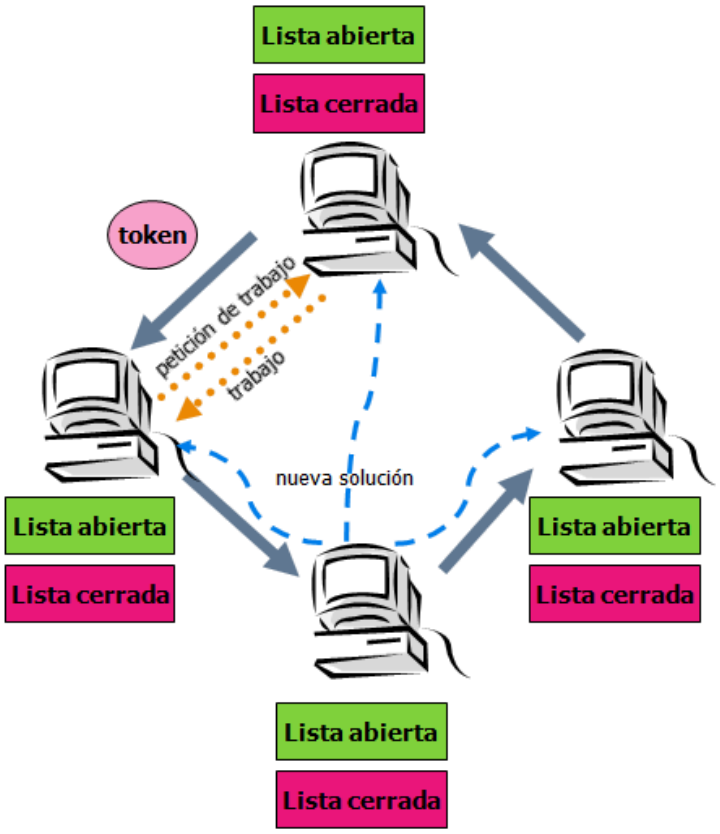

Figura 6.9: Esquema de solución paralela del problema del Puzzle.

La figura 6.9 muestra de forma gráfica el esquema de solución planteada, mostrando la distribución de las listas abierta y cerrada, así como también los tipos de comunicación, que pueden ser por:

- ✓ Requerimiento de trabajo y envío de trabajo.
- ✓ Aviso de nueva solución parcial encontrada.
- ✓ Pasaje del token de terminación.

En la figura 6.10 se visualiza el pseudocódigo del algoritmo paralelo estudiado anteriormente.

Entrada:

LW

Nombre archivo tablero inicial

Nombre archivo tablero final

// Inicialización

si soy el proceso 0

Leer tablero inicial (**ti**)

Leer tablero final (**tf**)

Insertar (lista abierta, **ti**, h(**ti**,**tf**))

Enviar tablero final a todos

Inicializar token

Autoenviarse el token

Sino

Recibir tablero final

//Inicialización ARR

donador = (myrank + 1)%cantworkers;

mientras (no terminar)

Nodos trabajados = 0

Mientras (mi lista abierta no este vacía) y (Nodos trabajados < **LW**)

Nodos trabajados ++

p = EliminarMínimo (lista abierta)

si (costo (p) < CMS)

si EsSolución(p)

Vaciar lista abierta

solución = p;

CMS = costo(p)

sino

hijos = Expandir(p)

Insertar(lista cerrada,p)

// Un nodo es aceptable si no está en la lista cerrada, o si está pero con un costo mayor al actual para cada hijo de n

Para cada nodo hijo en hijos

si (EsAceptable(hijo))

Insertar(lista abierta, hijo, h(hijo) + g(p) + 1)

Sino

Vaciar lista abierta

// Recibo valores de nuevas soluciones de costo minimo, actualizo mi CMS si es necesario. En ese caso, chequeo si el costo recibido es mayor al costo del nodo mínimo en lista abierta. Caso contrario se vacía la lista abierta.

si (lista abierta no vacía)

//Atiendo todos los pedidos de trabajo pendientes

//Cada vez que envía trabajo, cuenta el mensaje para garantizar terminación (contmsg) y pone su estado en NEGRO

// Si atendiendo se queda sin nodos para repartir, envía "mensaje de rechazo"

sino

```
                si encontró una nueva solución
                        Enviar a todos el costo de la nueva solución
                Pedir trabajo a mi donador
                // Mientras el worker espera la respuesta, atiende todo tipo de mensajes, sin orden determinado
                Mientras (lista abierta vacía) y (no terminar)
                        Esperar un mensaje
                        Si mensaje de trabajo
                                Recibe el trabajo
                                Resta su contmsg (para el algoritmo de terminación)
                                Inserta el trabajo recibido en la lista abierta
                                Setea estado en negro
                                Actualizar donador
                        Sino si mensaje de requerimiento de trabajo
                                Envía mensaje de rechazo
                        Sino si mensaje de rechazo
                                Actualizar donador
                                Pedir trabajo a otro donador
                        Sino si mensaje de token
                                Si CMS < costo(token)
                                        Si este proceso había encontrado la solución
                                                Actualiza costo token
                                                Actualiza la solución del token
                                Sino
                                        CMS =  costo(token)
                                Si soy proceso 0
                                        //Chequea condiciones de terminación
                                        Si  (token indica terminación)
                                                Terminar
                                        Sino
                                                Inicia nueva vuelta de token
                                Sino
                                        Actualizar color y contador de mensajes del token
                                        Pasar token al próximo worker
                                        estado   = BLANCO;
                        sino si mensaje de fin
                                Terminar
                        sino si mensaje nueva solución
                                Si costo(solución recibida) < CMS
                                        CMS = costo(solución recibida)
Si soy el proceso 0
        Avisar terminación a los demás procesos
```

Figura 6.10: Pseudocódigo de la solución paralela del problema del Puzzle.

6.3.4 Otras consideraciones

Con el fin de entender mejor el algoritmo paralelo, hay que realizar una serie de aclaraciones:

- ✓ En los algoritmos de búsqueda paralela, el speedup puede variar de una ejecución a otra, ya que el espacio de búsqueda examinado por los procesadores es repartido dinámicamente, y puede cambiar de una ejecución a otra.
- ✓ En el problema del Puzzle, pueden haber distintas soluciones (secuencias de movimientos que resuelven el tablero inicial en el tablero final) de igual costo, por lo tanto la solución encontrada por el algoritmo paralelo puede ser distinta a la encontrada por el algoritmo secuencial, y también puede diferir de la solución encontrada por otra ejecución del mismo algoritmo paralelo.

6.4 Implementación: discusión sobre las técnicas utilizadas y otras posibilidades

Tanto el algoritmo secuencial como el paralelo se implementaron usando el lenguaje C. El algoritmo paralelo utiliza la biblioteca de pasaje de mensajes MPI, usando en su mayoría comunicaciones asincrónicas.

Desde un principio se descartó la idea de una única lista abierta centralizada procurando evitar la contención que se generaría en el proceso que gestionaría la misma. Al principio solo el proceso 0 tendrá trabajo, a medida que comienza a progresar el algoritmo los demás procesadores irán pidiendo trabajo siguiendo la técnica de balance de carga ARR.

Dado que la lista abierta ahora está distribuida, para evitar que cada proceso del algoritmo paralelo procese mayor cantidad de nodos que el algoritmo secuencial, la estrategia de división de trabajo garantiza que todos los procesos tengan algunos de los mejores nodos del sistema.

Diversos algoritmos de ajuste de calidad de nodos han sido estudiados, en particular se podrían mover nodos entre las listas abiertas cada cierto tiempo, implementando una comunicación en anillo, random, o centralizando nodos en una estructura global.

Por otra parte, se optó por que cada proceso chequee localmente por ciclos. Esto podría llevar a que dos procesos puedan estar procesando en algún momento nodos iguales. Una alternativa es realizar un hashing de nodos procesados entre las listas cerradas de los procesos: cuando un proceso expande un nodo obtiene, aplicándole una función de hash, el número del procesador cuya lista cerrada debería contener el nodo, si fue procesado. Ese proceso chequea si el nodo está o no en su lista cerrada y retorna el resultado. La elección de una y otra estrategia se podría basar en un análisis del

overhead de comunicaciones de la última estrategia, frente al procesamiento extra de la primera.

7. Trabajo experimental

En esta sección se estudian las métricas usadas comúnmente para evaluar el rendimiento de un programa paralelo (speedup y eficiencia), junto con el concepto de escalabilidad de un sistema paralelo. Además se exponen las causas de overhead más comunes en un algoritmo paralelo.

A continuación se comentan las pruebas secuenciales realizadas, variando la heurística para la valuación de costo de los nodos, y se discuten los resultados alcanzados.

Posteriormente, se exponen las características del cluster utilizado, y se muestran los resultados paralelos obtenidos, analizando la superlinealidad alcanzada y eficiencia. Asimismo se realiza un análisis de la escalabilidad de la aplicación.

7.1 Métricas

En general, un algoritmo secuencial se evalúa en términos de su tiempo de ejecución, el cual dependerá del tamaño de su entrada. El comportamiento asintótico del tiempo de ejecución será igual en cualquier plataforma secuencial.

Sin embargo, la performance de un algoritmo paralelo no solo dependerá del tamaño del problema, sino que dependerá de la arquitectura (cantidad de procesadores, características del medio de comunicación, entre otros), distribución de los procesos en procesadores, técnica para balancear la carga utilizada, etc.

En el procesamiento paralelo, es de interés evaluar el desempeño de un sistema paralelo y así poder compararlo con otro. Existen diversas métricas para evaluar la performance de un sistema paralelo, entre las más conocidas se encuentran el tiempo paralelo, el speedup y la eficiencia.

7.1.1 Speedup

El *factor de Speedup absoluto Sp* mide la ganancia obtenida al resolver un problema en paralelo, con respecto a su implementación secuencial, y se define como la relación entre el tiempo de ejecución del mejor algoritmo secuencial sobre una máquina monoprocesador y el tiempo de ejecución del algoritmo paralelo sobre una máquina multiprocesador [GRA03] [LEO01]. La figura 7.1 muestra la fórmula para calcular el speedup.

$$S_p = \frac{T_s}{T_p}$$

Figura 7.1: Speedup

El speedup normalmente se trata de maximizar en el desarrollo de aplicaciones paralelas, y estará limitado por el máximo grado de concurrencia que puede obtenerse de la aplicación, por el inevitable componente secuencial del algoritmo y por el número de procesadores N disponibles para la ejecución. [QUI93]

En general, en un sistema paralelo con N máquinas se esperaría obtener una aceleración máxima de N, por eso se dice que el Speedup teóricamente varía entre 1 y N, pero puede ser mejorada en algunos casos dando lugar al concepto de *Superlinealidad Su.* Analizar el por qué ocurre esta anomalía es de gran interés dentro del campo del paralelismo, en particular en los problemas de optimización discreta.

En dichos problemas, la exploración del espacio total de soluciones posibles puede reducirse al distribuir el trabajo entre N procesadores y poder "cortar" o "finalizar" la búsqueda global al llegar al resultado esperado en cualquiera de ellos [HEL90] [MAN02] . Es decir que en teoría la arquitectura de clúster podrá permitirnos alcanzar superlinealidad, dependiendo del balance de carga, la heterogeneidad de los procesadores y la relación tiempo de procesamiento/tiempo de comunicaciones del algoritmo empleado [SAN07].

7.1.2 Eficiencia

La Eficiencia mide el uso efectivo de los recursos de cómputo y resulta una métrica de calidad y de costo del algoritmo paralelo.

Se define como *Eficiencia* la relación entre el Speedup y el Speedup óptimo. En el caso de un clúster homogéneo, el Speedup óptimo es N, el número de procesadores utilizados. La figura 7.2 muestra la fórmula para calcular la eficiencia.

$$E = \frac{S_p}{N}$$

Figura 7.2: Eficiencia

Esta definición pone la Eficiencia entre 0 y 1. Alcanzar valores cercanos a 1 significa que se logra *Sp* cercano al óptimo N. La eficiencia no siempre se puede mantener al escalar los problemas, al incrementar el número de procesadores o al portar el algoritmo sobre otra arquitectura multiprocesador [BUY99].

Llamaremos *Pseudoeficiencia* a aquellos casos donde la eficiencia es mayor a 1. La Pseudoeficiencia puede aparecer en algunos casos donde se obtiene superlinealidad en la ejecución del algoritmo paralelo. Esta eficiencia no es realista, ya que indicaría que cada procesador está siendo utilizado más de un 100%.

7.2 Escalabilidad

El algoritmo de búsqueda paralelo presentado es utilizado para resolver problemas de gran escala, por lo tanto se debe evaluar su rendimiento a medida que crece el tamaño del problema. También al aumentar el problema es probable que el usuario quiera utilizar una arquitectura más potente, de modo de obtener un tiempo de respuesta menor.

La escalabilidad es una propiedad deseable para un sistema paralelo. Un sistema paralelo normalmente escala incrementando la cantidad de procesadores o la potencia de los mismos, es decir se escala su arquitectura, o incrementando el volumen de datos a procesar, es decir escala el problema. Para el caso de estudio propuesto, escalar el problema significa aumentar el tamaño del tablero (N), lo cual causará un crecimiento importante del espacio de estados.

Es de interés realizar un estudio de la escalabilidad de un sistema que permita al usuario conocer el rendimiento del mismo a medida que escala la arquitectura y el problema, en especial para los problemas de optimización discreta.

La eficiencia es una métrica comúnmente utilizada para estudiar la escalabilidad. Diversos autores han propuesto métodos para evaluar la escalabilidad de los sistemas paralelos. Un método representativo para arquitecturas homogéneas es el análisis de isoeficiencia, el cual plantea que un sistema es escalable si es posible mantener la eficiencia del sistema en un valor fijo al aumentar el número de procesadores y tamaño del problema. La *función de isoeficiencia* indica cuánto tiene que aumentar el tamaño del problema para poder incluir más procesadores sin que la eficiencia del sistema se vea afectada. [GRA03]

El modelo de isoeficiencia se basa en las siguientes consideraciones:

- ✓ Para un problema de tamaño determinado, a medida que se incrementa el número de procesadores, la eficiencia del sistema paralelo decae.
- ✓ En muchos casos, la eficiencia aumenta al escalar el tamaño del problema manteniendo el número de procesadores constante.

Diversos factores podrán hacer decaer la eficiencia a medida que se escala el sistema paralelo. En la siguiente sección se estudian las causas de overhead asociados al paralelismo.

7.3 Causas de overhead en los algoritmos paralelos

En un algoritmo paralelo existen diversas fuentes de overhead que disminuyen su rendimiento óptimo. Se destacan las siguientes causas:

- ✓ Comunicaciones entre procesadores.
- ✓ Procesadores ociosos: desbalance de carga, sincronización, componentes secuenciales.
- ✓ Exceso de cómputo en el algoritmo paralelo: el algoritmo secuencial más rápido puede ser difícil o imposible de paralelizar, teniendo que implementar un algoritmo paralelo en base a una técnica deficiente. También en algunos algoritmos paralelos basados en técnicas óptimas, puede existir exceso de cálculo debido a que algunos resultados, que en el algoritmo secuencial podían ser reusados, no puedan ser reusados en el algoritmo paralelo debido a que están siendo generados por distintos procesadores, por lo que se deben calcular múltiples veces.

Al desarrollar una aplicación paralela se deben tener en cuenta estos factores con el objetivo de minimizarlos.

En el algoritmo paralelo presentado para resolver el problema del Puzzle N^2-1, las comunicaciones entre procesadores se redujeron lo máximo posible. Solo existen tres circunstancias por las cuales los procesadores deben comunicarse necesariamente:

- ✓ Balance de carga: esta comunicación es costosa cuando se transmite la carga de trabajo de un procesador a otro.
- ✓ Comunicación de solución optima encontrada: al comunicar solo el costo de la solución, no hay un costo alto debido al traslado de información.
- ✓ Terminación: comunicación no costosa ya que sólo se pasa un token con poca información, y además el procesador que tiene el token y lo pasa es porque en ese momento estaba ocioso.

Con el fin de disminuir el tiempo ocioso de un procesador por no disponer de trabajo para realizar, se implementó un algoritmo de balance de carga dinámico distribuido. Asimismo, las comunicaciones utilizadas son en su mayoría asincrónicas. Las comunicaciones sincrónicas se utilizaron en casos donde se sabe que el otro proceso estará pendiente para recibir el mensaje (no está trabajando). Todos los procesadores realizan el mismo trabajo, por lo tanto no hay códigos serie en este algoritmo.

Para finalizar, el algoritmo usado para la paralelización es óptimo, pero debido a que los procesadores trabajan sobre distintos nodos en forma concurrente, puede ocurrir que se trabaje sobre nodos que el algoritmo secuencial no procesa.

7.4 Resultados Secuenciales

La predicción de la performance de los algoritmos dependientes de los datos es de gran interés ya que para entradas de tamaño similar puede haber gran variabilidad en los tiempos de ejecución. Es de interés descubrir los factores que influyen en el rendimiento del algoritmo secuencial y paralelo, en especial para el problema del Puzzle.

A los efectos de comprobar la mejora o no de los tiempos secuenciales a medida que se afina la heurística para valuar el costo de los nodos durante la búsqueda, se realizaron pruebas con aproximadamente 60 configuraciones iniciales, siendo N=5. Las heurísticas utilizadas son las siguientes:

- ✓ Suma de las Distancias de Manhattan (*H1*)
- ✓ H1 + Detección de Conflictos Lineales (*H2*)
- ✓ H2 + Últimos movimientos (*H3*)
- ✓ H3 + Esquinas del Tablero (*H4*)

Las configuraciones iniciales se crearon a partir de aplicar un número al azar de movimientos legales hacia atrás a cada uno de los patrones de tableros definidos en 6.5. En cada iteración, el movimiento a realizar también fue seleccionado en forma aleatoria.[15]

Para evaluar algunos aspectos de interés del algoritmo, se calcularon: el número de nodos procesados[16], explotados[17] y podados[18], junto con el tamaño promedio de las listas de la estructura utilizada como lista cerrada.

7.4.1 Evidencia de dependencia de los datos

A modo de ejemplo, se muestran algunos resultados secuenciales para demostrar que el algoritmo que resuelve el problema del Puzzle N^2-1 es dependiente de los datos de

[15] Las fichas no fueron colocadas en los tableros en forma *random* ya que el tablero generado poseía un grado de desorden excesivamente grande, lo que hacía imposible su resolución en una máquina de las características comentadas en 7.5, dado que se agotaban los recursos antes de poder encontrar una solución.

[16] Un nodo *procesado* es un nodo del grafo elegido en una iteración del algoritmo para ser insertado en la lista cerrada y a partir de él generar sus sucesores.

[17] Un nodo *explotado* es aquel que fue generado a partir de aplicar movimientos legales a un nodo.

[18] Un nodo *podado* es un nodo que es descartado por haberse procesado anteriormente o, en el algoritmo paralelo, por tener costo mayor o igual a la mejor solución alcanzada hasta el momento.

entrada. Podemos observar que, en general, los tiempos de ejecución del algoritmo varían, aún cuando se mantiene el tamaño del tablero, heurística y solución a alcanzar.

La tabla 7.1 muestra un subconjunto de las pruebas secuenciales, para tableros con N=5, utilizando como tablero solución el tablero clásico y la heurística SDM.

Tabla 7.1: Prueba de dependencia de los datos. N=5, SDM, tablero solución clásico.

Nro. Prueba	Desorden Inicial	Pasos solución	Tiempo
1	7	31	857,6
2	24	36	81,6
3	20	34	53,6
4	30	42	92,8
5	20	34	104,2
6	21	35	17,4
7	25	37	285,5
8	25	41	6367,7
9	22	34	129,96
10	22	38	746,5
11	23	39	3311,4
12	36	44	212
13	28	38	79,9
14	34	42	73,7
15	32	44	774,3
16	38	46	1157,3
17	22	38	367,9
18	20	36	1604,1
19	30	42	53,6
20	24	36	119,57
21	32	44	311,61
22	32	44	253,47
23	32	46	24178,9
24	28	38	129,1
25	26	36	207,5

Resultados similares se obtuvieron con las configuraciones finales de la figura 6.4. En la tabla 7.2 se visualizan algunos resultados con N=5, heurística SDM y tablero solución 1. Evidencias parecidas, variando el tamaño del tablero y manteniendo la heurística y la configuración final se pueden encontrar en estudios previos [SAN07].

Tabla 7.2: Prueba de dependencia de los datos. N=5, SDM, tablero solución 6.4.a.

Nro. Prueba	Desorden Inicial	Pasos solución	Tiempo
26	28	40	240,15
27	30	42	1979,7
28	36	46	414,3
29	26	38	331,3
30	36	48	2038,4
31	38	48	885,23
32	20	36	322,3
33	24	40	1855,54
34	30	42	957,22
35	36	46	520,87
36	32	46	1887,37
37	32	42	284,7
38	32	40	53,2
39	36	46	64,9

De los resultados exhibidos se puede observar que el tiempo no tiene una dependencia directa respecto de la diferencia entre la cantidad de pasos y el desorden inicial. Por ejemplo, las pruebas número 2, 4, 7, 15, 20, 21, 22 tienen igual distancia entre el tablero solución (a saber, 12), sin embargo los tiempos para su resolución son muy variantes.

A partir de ordenar las pruebas de la tabla 7.1. por cantidad de nodos procesados de mayor a menor, se obtiene la tabla 7.3, en la cual se agregó información adicional obtenida durante la búsqueda (nodos procesados, expandidos y podados, y longitud promedio de las listas de la estructura utilizada como lista cerrada *LPLC*). Este último factor es muy dependiente del grafo que se genera durante la búsqueda.

Como se puede ver, los tiempos obtenidos están claramente relacionados con la cantidad de nodos procesados, es decir a medida que decrece la cantidad de nodos procesados en general también decaen los tiempos. Ahora bien, es interesante descubrir los factores que causan las excepciones (pruebas número 18, 1, 17, 7, 9, 5, 13 y 2). En dichos casos, el valor LPCL aumentó considerablemente con respecto a las pruebas que las anteceden en el orden, y en algunos casos también lo hizo la cantidad de nodos expandidos.

De modo similar, se puede aplicar la misma estrategia a los resultados de la tabla 7.2.

Tabla 7.3: Tiempos de la tabla 7.2 ordenados por "Nodos Procesados"

Nro. Prueba	Desorden Inicial	Pasos solución	Nodos Procesados	Nodos Expandidos	Nodos Podados	LPLC	Tiempo
23	32	46	1068261	3542300	1200867	27391	24178,9
8	25	41	446360	1484298	493125	13526	6367,7
11	23	39	350515	1147890	382407	11306	3311,4
16	38	46	238915	790510	253504	5827	1157,3
18	20	36	226430	753597	245193	8086	1604,1
15	32	44	197620	652687	213355	5200	774,3
10	22	38	164916	539347	177881	5497	746,5
1	7	31	135371	451601	150477	7124	857,6
21	32	44	124368	410696	134621	3272	311,61
22	32	44	114370	381113	121582	3009	253,47
17	22	38	110227	369903	119730	3674	367,9
12	36	44	109275	359177	115047	2731	212
7	25	37	108390	362463	118226	3496	285,5
25	26	36	89883	300146	96552	2899	207,5
20	24	36	77843	254609	86366	2594	119,57
9	22	34	72058	239809	79402	2573	129,96
24	28	38	68090	228516	72221	2063	129,1
14	34	42	66697	218081	70043	1755	73,7
5	20	34	63228	209953	68513	2341	104,2
4	30	42	61442	201784	64803	1755	92,8
13	28	38	59607	200422	64409	1806	79,9
19	30	42	51784	172902	54849	1438	53,6
2	24	36	49710	197161	66226	1990	81,6
3	20	34	48769	160268	52253	1806	53,6
6	21	35	29365	97847	30768	1048	17,4

7.4.2 Pruebas variando la heurística

A continuación se muestra un subconjunto de las pruebas realizadas, variando la función heurística para estimar los costos de los nodos (H1, H2, H3, H4) y configuración inicial y final. Para todos los casos, N = 5.

Tabla 7.4: Pruebas secuenciales con distintas heurísticas

Nro. Prueba	Tablero Solución	Tiempos H1	Tiempos H2	Tiempos H3	Tiempos H4
1	3	2019,68	61,83	268,1	259,92
2	1	885,23	50,73	8,4	0,64
3	4	119,57	3,23	0,87	1,15
4	3	7687,9	276,63	45,53	0,77
5	1	322,3	4,65	3,77	3,8
6	2	1485,5	25,14	17,63	55,2
7	1	1855,54	17,58	24,9	4,8
8	3	1319,6	361,94	269,99	11,27
9	4	311,61	0,46	0,25	0,057
10	2	1185,96	76,13	103,2	11,01
11	1	957,22	310,31	1245,71	416,39
12	4	253,47	3,3	13,57	2,23
13	4	24178,9	1460,8	1681,09	251,8
14	3	2894	26,1	12,7	12,4
15	1	520,87	104,93	122,27	12,05
16	1	1887,37	15,11	3,73	3,76
17	3	1553,9	57,12	2,69	0,99
18	2	106,1	3,1	3,77	0,68
19	4	129,1	2,97	3,06	2,83
20	1	284,7	23,14	15,65	0,47
21	4	207,5	31,43	19,31	17,5
22	1	53,2	2,47	2,68	2,76
23	1	64,9	4,13	2,99	3,82
24	3	20	1,18	0,32	0,07
25	3	334,4	7,66	52,58	0,74

En general, se ha notado que al afinar la heurística disminuye el tiempo de ejecución, cantidad de nodos procesados y explotados, y el factor LPLC. Sobre el total de las pruebas, se calculó para cada heurística la cantidad de pruebas para las cuales dicha función mejora en tiempos a las restantes. A partir de dichos datos se calcularon los porcentajes, que se muestran en la figura 7.3. Se puede observar que en la mayoría de los casos (65%) la heurística H4 produjo los mejores tiempos de ejecución, seguida por H3 (20%) y H2 (15%). Se puede notar que H1 no fue mejor en tiempos en ninguna de las pruebas.

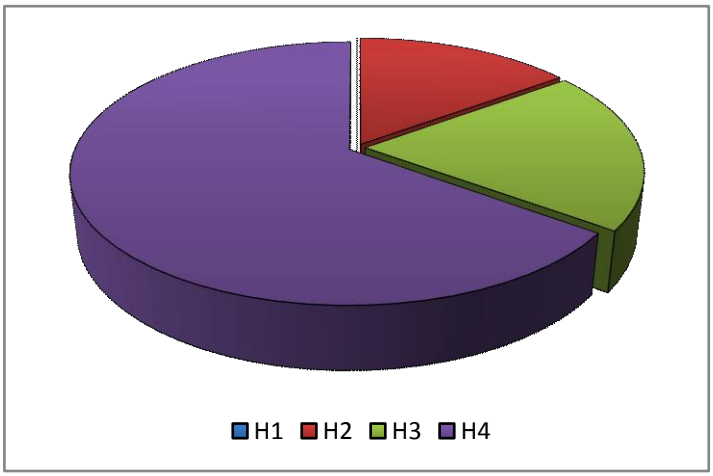

Figura 7.3: Porcentaje de elección de heurísticas

Al afinar la función heurística cada nodo es valuado de manera más exacta. De lo anterior, se podría pensar que es imposible que una heurística más afinada pueda procesar mayor cantidad de nodos, o tardar más tiempo, para resolver un tablero inicial. Esto no es cierto, según se visualiza en la tabla 7.4. y en los porcentajes anteriores. Por ejemplo, notar que para las pruebas 1, 11 y 22 la heurística que minimiza los tiempos es H2, y H3 minimiza los tiempos para las pruebas 3, 5, 6, 16 y 23.

Para entender lo anterior, se deben hacer las siguientes consideraciones:

- ✓ A medida que se mejora la heurística, es mayor el tiempo de procesamiento por nodo. En consecuencia, puede ocurrir que para un tablero inicial y dos heurísticas h1 y h2, donde h2 mejora a h1, el algoritmo procese similar cantidad de nodos o incluso que con h2 procese menor cantidad de nodos, y decrezca el factor LPLC y la cantidad de nodos explotados, y sin embargo tardar más en tiempo. Se observó dicha situación, por ejemplo, en la prueba 23.
- ✓ Al mejorar la heurística podría ocurrir que se incremente el factor LPLC, dado que cambia la valuación de los nodos, y los mismos se ordenarán de forma distinta en las listas de la estructura utilizada como lista cerrada.
- ✓ Al mejorar la heurística podría ocurrir que se procese mayor cantidad de nodos, debido a los nodos de coste igual que la solución. [RUS03].

Como conclusión se puede decir que, al mejorar la heurística es probable que se procesen menor cantidad de nodos, pero será mayor el tiempo de cómputo. El uso de heurísticas más potentes combinadas con el procesamiento paralelo permitirá resolver instancias más grandes del problema.

7.5 Resultados Paralelos

Para las pruebas se utilizó el clúster homogéneo IBM de la Facultad de Informática, compuesto por 20 máquinas, conectadas por una red LAN de 100 Mbits, cada una con las siguientes características:

- ✓ Procesador Pentium IV de 2.4GHz.
- ✓ Cache de 512 KB.
- ✓ 1GB de memoria RAM.
- ✓ 40 GB de disco rígido.
- ✓ Sistema operativo: Linux Fedora 8.

El entorno MPI utilizado es la versión de LAM 7.1.4.

Al igual que en el algoritmo secuencial, para medir el tiempo de ejecución se utilizó la función MPI_Wtime de la librería MPI. Dicha función no requiere inicializar el entorno MPI para que la misma funcione, por lo que no añade overhead adicional en los tiempos secuenciales.

A continuación se exponen algunos resultados obtenidos que muestran claramente la superlinealidad en las búsquedas paralelas en grafos y se comenta las razones por las cuales se da esta situación. En todos los casos, el speedup se calculó teniendo en cuenta el mejor tiempo secuencial obtenido para resolver el tablero inicial en el tablero final particular (*speedup absoluto*), mediante la ejecución del algoritmo secuencial con las distintas heurísticas.

Posteriormente se realiza un análisis del parámetro LW, utilizado en la búsqueda paralela. La predicción del parámetro LW óptimo, es decir tal que minimice los tiempos o nodos procesados, es un aporte y de gran interés.

Para finalizar, se exhiben pruebas realizadas sobre tableros de estructura similar, variando el N y la cantidad de máquinas, para analizar la escalabilidad del algoritmo. El análisis de la escalabilidad del problema y arquitectura es otra contribución de este trabajo, ya que predecir cuál es la cantidad óptima de máquinas para un tablero inicial dado causaría una utilización más eficiente de los recursos disponibles.

7.5.1 Evidencia de superlinealidad

Con el fin de analizar la existencia de superlinealidad en los algoritmos de búsqueda paralela en grafos, se realizaron pruebas con distintas configuraciones iniciales y finales, con 4 máquinas y con tableros con N = 5, variando el parámetro LW.

En particular, para cada configuración inicial se ejecutó el algoritmo secuencial con las distintas heurísticas. La tabla 7.5 muestra un subconjunto de los resultados paralelos

para dichas configuraciones con la heurística que mejoraba el tiempo secuencial (ya que el speedup se debe tomar respecto al mejor algoritmo secuencial) y el LW óptimo. Hasta este punto, se espera que el algoritmo paralelo con la misma heurística sea el que mejore los tiempos. El número de prueba identifica el tablero inicial y final.

Los resultados obtenidos revelan claros indicios de superlinealidad en la ejecución. En la figura 7.4 se muestra el porcentaje total de resultados superlineales obtenidos. Tomando en cuenta el total de las pruebas, en el 65 % se obtuvieron resultados superlineales.

Tabla 7.5: Pruebas paralelas que indican existencia de superlinealidad

Nro. prueba	Heurística	Bloque (LW)	Speedup	Eficiencia/ Pseudoeficiencia
1	H2	1000	17,32	4,33
2	H2	1000	2,09	0,52
3	H2	1000	16,86	4,21
4	H2	1000	12,49	3,12
5	H2	1000	6,04	1,51
6	H2	750	12,22	3,05
10	H3	1000	9,20	2,30
11	H3	750	7,95	1,99
12	H3	750	2,44	0,61
13	H3	500	2,22	0,56
14	H3	1000	5,47	1,37
15	H3	1000	4,58	1,14
22	H4	1000	15,26	3,82
23	H4	1000	6,60	1,65
24	H4	1000	6,67	1,67
25	H4	750	0,94	0,23
26	H4	750	21,16	5,29
27	H4	750	9,09	2,27
28	H4	750	1,59	0,40
29	H4	500	12,10	3,02

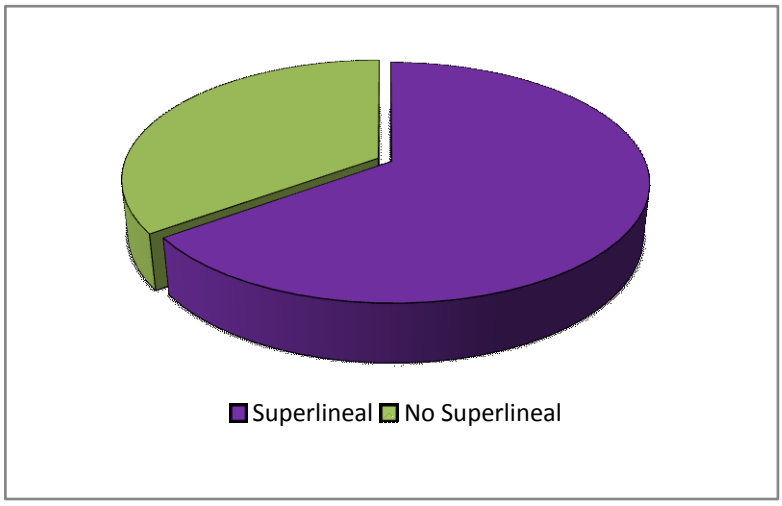

Figura 7.4: Porcentaje de pruebas superlineales

Resulta interesante discutir acerca de los posibles causantes de la superlinealidad obtenida en la experimentación.

En general, en los problemas de búsqueda en grafos es posible obtener superlinealidad. Uno de los causantes principales se da cuando en la ejecución paralela se llega al nodo solución luego de examinar menor cantidad de nodos que el algoritmo secuencial, siempre y cuando el overhead de comunicación no sea alto. En los algoritmos Best First Search, dicha anomalía es causada por nodos que tienen idéntico costo.

Para explicar lo expuesto anteriormente, se plantea la siguiente situación: supongamos algoritmo ha llegado a una instancia donde hay gran cantidad de nodos con costo x (mínimo) y ninguno es la solución. También supongamos que la solución tiene costo x + 1. El algoritmo secuencial A*, antes de tomar algún nodo con costo x+1 deberá procesar todos aquellos nodos con costo x[19], lo cual puede tomar gran cantidad de tiempo.

En el algoritmo paralelo A*, los nodos del espacio de estados están distribuidos en las listas abiertas de los procesadores, además la técnica de balance de carga implementada hace que el donador de trabajo intercambie parte de sus mejores y peores nodos de su lista abierta con el procesador ocioso que le solicitó trabajo. Esto ayuda que los nodos con igual costo se distribuyan entre los procesadores. Es más, la técnica de balance ayuda a distribuir los mejores nodos entre los procesadores. Por lo tanto, es probable que un procesador tome el nodo solución antes de que todos los nodos de costo x sean procesados.

Al mismo tiempo, los procesadores pueden encontrar soluciones sub-óptimas que, al comunicarlas, permiten podar o cortar ramas que llevan a soluciones no óptimas en

[19] Recordar que el algoritmo A* en cada iteración toma de la lista abierta el nodo de menor costo.

otros procesadores. Al encontrar la solución mínima en uno de los procesadores, se termina la búsqueda.

Así, en algunos casos, se ha notado una reducción en la cantidad de nodos que procesa el algoritmo paralelo, en comparación con el algoritmo secuencial, acompañado de un aumento en la cantidad de nodos podados.

Otro causante de la superlinealidad es la reducción en cada procesador del tamaño de las listas en la estructura utilizada como lista cerrada. Al distribuir el trabajo entre N procesadores, cada lista cerrada es de menor tamaño y las búsquedas para la eliminación de ciclos son más rápidas, haciendo más probable que quepan en memoria.

A los efectos de comprobar las causas planteadas anteriormente, se calculó el porcentaje de reducción de nodos procesados y expandidos, y el porcentaje de incremento de nodos podados, junto con el porcentaje de reducción de las listas de la estructura planteada como lista cerrada (en el algoritmo paralelo, se tomó la longitud promedio máxima entre todos los nodos).

Se notó que los casos con mayor nivel de superlinealidad presentaron, en su mayoría, una reducción de la cantidad de nodos totales procesados entre el 1% y 65%, y un incremento de nodos podados entre el 1% y 103%. En los casos donde no se redujo la cantidad de nodos procesados, hubo un fuerte incremento de la cantidad de nodos podados, y viceversa, lo que amortiguó el incremento en el speedup.

En la tabla 7.6 se muestran los 10 mejores speedup obtenidos en las pruebas. El número de prueba identifica la configuración inicial, final y la heurística, junto con el valor de LW usado.

Tabla 7.6. Diez mejores Speedup

Nro. prueba	Speedup	% Reducción de nodos procesados	% Incremento de nodos podados	% Reducción de listas
20	21,24	8	53	70
26	21,16	11	38	73
31	22,36	1	86	67
32	18,54	-4	98	70
33	86,30	65	-43	89
34	34,25	33	6	77
35	25,95	11	70	74
50	19,75	43	12	81
54	21,27	39	1	78
57	18,04	-10	103	66

En general, las pruebas cuyos speedup fueron superlineales redujeron la longitud de la listas entre un 40% y 89%, y se incrementó la poda de nodos entre un 1% y un 255% (posibilitado en algunos casos por un incremento en la cantidad de nodos explotados).

7.5.2 Elección de Heurística

En la sección anterior se supuso que la heurística que mejoraba los tiempos secuenciales también lo haría con los tiempos paralelos. A partir de ejecutar las mismas pruebas pero variando la heurística, se confirmó que en algunos casos esto no es así.

En la tabla 7.7. muestra las pruebas en las cuales la heurística H4 mejoró significativamente los tiempos en comparación a la heurística elegida (H2 o H3). Otros casos no se han tenido en cuenta, debido a que la mejora era muy pobre.

Tabla 7.7: Error de la elección de heurística

	HEURISTICA ELEGIDA			HEURISTICA H4	
Nro. prueba	Heurística	Bloque (LW)	Tiempo	Bloque (LW)	Tiempo
5	H2	1000	36,54	500	11,26
9	H2	500	106,6	1000	26,72
18	H3	500	7,31	1000	0,68

7.5.3 Parámetro LW

El parámetro LW, presentado en la sección 6, es una medida de la cantidad de nodos que cada procesador trabaja por iteración del algoritmo. Posteriormente, el procesador hace el chequeo de peticiones de trabajo hechas por otros procesadores y también examina soluciones óptimas encontradas por otros procesadores.

Un valor grande de LW podría hacer que los procesadores ociosos a espera de trabajo permanezcan en dicho estado por un largo tiempo. También, al no recibir los costos de soluciones sub-óptimas encontradas por otros, podrían procesar nodos innecesarios.

Un valor muy chico de LW permitiría chequeos más frecuentes, pero podría causar un aumento en los tiempos debido a las constantes verificaciones.

Se observó en las pruebas que no existe un valor de LW óptimo, sino que éste depende de diferentes factores. El valor de LW óptimo dependerá de cómo se distribuya el espacio de estados dinámicamente. Por ejemplo, si los procesadores tienden a quedarse sin trabajo, convendrá un valor chico de LW, así los procesadores seleccionados como donadores podrán atender al procesador ocioso.

7.5.4 Escalabilidad

Para observar la forma en que el algoritmo escala al aumentar el tamaño del problema (N=4,5,6), se definieron dos tipos de configuración inicial:

- ✓ *Configuración 1:* invertir en el subtablero de 4x4 inferior derecho la tercera columna y luego la tercera fila.
- ✓ *Configuración 2:* invertir en el subtablero de 4x4 inferior derecho la segunda columna y luego la segunda fila.

La solución buscada será respecto al tablero clásico.

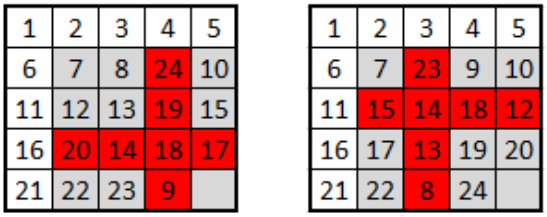

Figura 7.5: Ejemplos para tableros de 5x5 **a.** Configuración 1 **b.** Configuración 2.

Además, con el fin de analizar el comportamiento de la aplicación al escalar la arquitectura, cada tablero mencionado se probó con P procesadores, donde P = {4, 8, 12, 16}.

Para cada configuración, y cada P se varío el tamaño de bloque LW, para buscar el valor que optimice los tiempos. Además cada prueba se realizó con las heurísticas H2, H3 y H4, descartando por completo la heurística H1.

La tabla 7.8 muestra los mejores tiempos secuenciales para ambas configuraciones, variando N, junto con la heurística que se utilizó para obtenerlos. Estos tiempos fueron utilizados para calcular el speedup.

Tabla 7.8: Tiempos secuenciales

N	Configuración	Heurística	Desorden inicial	Solución	Tiempo
4	1	CT	20	40	488,77
4	2	LC	20	40	457,8
5	1	LC	20	40	1620,5
5	2	LC	20	40	1916,9
6	1	LC	20	40	1617,53
6	2	LC	20	40	1791,9

La tabla 7.8 muestra los tiempos paralelos obtenidos con la configuración 1, variando N, la heurística y P, junto con el LW óptimo. De la misma manera, la tabla 7.9 muestra

los tiempos paralelos de la configuración 2. Terminando con un resumen de los tiempos en las tablas 7.10 y 7.11, respectivamente para la configuración 1 y 2, con los mejores tiempos para cada N y configuración de cluster (P).

Tabla 7.8: Tiempos paralelos de la configuración 1.

Tablero	Heurística	Cantidad de procesadores	LW	Speedup	Pseudoeficiencia
4x4	H2	4	1000	10,33	2,58
		8	750	16,16	2,02
		12	750	24,56	2,05
		16	500	49,37	3,09
	H3	4	1000	10,47	2,62
		8	1000	31,80	3,98
		12	1000	46,11	3,84
		16	1000	72,95	4,56
	H4	4	1000	16,83	4,21
		8	1000	21,53	2,69
		12	750	45,01	3,75
		16	1000	47,68	2,98
5x5	H2	4	500	14,03	3,51
		8	750	33,16	4,14
		12	750	46,57	3,88
		16	750	68,96	4,31
	H3	4	500	9,99	2,50
		8	500	22,31	2,79
		12	500	30,78	2,56
		16	500	57,75	3,61
	H4	4	500	13,45	3,36
		8	500	23,76	2,97
		12	500	32,08	2,67
		16	500	59,14	3,70
6x6	H2	4	1000	6,92	1,73
		8	750	22,29	2,79
		12	500	25,66	2,14
		16	750	40,49	2,53
	H3	4	1000	10,02	2,50
		8	750	26,37	3,30
		12	750	39,74	3,31
		16	750	49,86	3,12
	H4	4	1000	10,02	2,50
		8	750	27,05	3,38
		12	750	38,79	3,23
		16	500	38,06	2,38

Tabla 7.10: Tiempos paralelos de la configuración 2.

Tablero	Heurística	Cantidad de procesadores	LW	Speedup	Pseudoeficiencia
4x4	H2	4	750	12,46	3,12
		8	1000	28,05	3,51
		12	1000	44,88	3,74
		16	500	68,43	4,28
	H3	4	1000	11,42	2,85
		8	750	21,84	2,73
		12	1000	35,35	2,95
		16	1000	59,92	3,75
	H4	4	500	10,74	2,69
		8	500	19,67	2,46
		12	500	29,35	2,45
		16	1000	37,93	2,37
5x5	H2	4	750	15,93	3,98
		8	750	42,50	5,31
		12	750	46,64	3,89
		16	500	78,88	4,93
	H3	4	750	16,73	4,18
		8	750	47,80	5,98
		12	750	73,61	6,13
		16	750	74,76	4,67
	H4	4	750	16,92	4,23
		8	750	47,24	5,90
		12	750	72,09	6,01
		16	750	72,20	4,51
6x6	H2	4	500	10,28	2,57
		8	500	21,76	2,72
		12	500	27,36	2,28
		16	500	37,02	2,31
	H3	4	750	3,71	0,93
		8	500	14,80	1,85
		12	750	27,41	2,28
		16	750	40,42	2,53
	H4	4	500	6,14	1,53
		8	500	14,79	1,85
		12	750	33,82	2,82
		16	750	46,54	2,91

Tabla 7.11: Mejores tiempos paralelos de la configuración 1.

Tablero	Cantidad de procesadores	Heurística	LW	Speedup	Pseudoeficiencia
4x4	4	CT	1000	16,83	4,21
	8	LM	1000	31,80	3,98
	12	LM	1000	46,11	3,84
	16	LM	1000	72,95	4,56
5x5	4	LC	500	14,03	3,51
	8	LC	750	33,16	4,14
	12	LC	750	46,57	3,88
	16	LC	750	68,96	4,31
6x6	4	LM	1000	10,02	2,50
	8	CT	750	27,05	3,38
	12	LM	750	39,74	3,31
	16	LM	750	49,86	3,12

Tabla 7.12: Mejores tiempos paralelos de la configuración 2.

Tablero	Cantidad de procesadores	Heurística	LW	Speedup	Pseudoeficiencia
4x4	4	LC	750	12,46	3,12
	8	LC	1000	28,05	3,51
	12	LC	1000	44,88	3,74
	16	LC	500	68,43	4,28
5x5	4	CT	750	16,92	4,23
	8	LM	750	47,80	5,98
	12	LC	750	46,64	3,89
	16	LC	500	78,88	4,93
6x6	4	LC	500	10,28	2,57
	8	LM	500	14,80	1,85
	12	CT	750	33,82	2,82
	16	CT	750	46,54	2,91

De los resultados anteriores se observa que al aumentar la cantidad de procesadores, para un N particular y una configuración particular, el speedup crece. Por ello, la eficiencia tiende a crecer y luego decae al aumentar la cantidad de maquinas, o se mantiene aproximadamente.

Se observó que, en general, al aumentar el tamaño del tablero disminuye el speedup. Esto ocurre tanto en la comparación de los resultados obtenidos a partir de una

heurística, como en el resumen de los mejores tiempos. Se destacan a continuación las posibles causas:

- ✓ El espacio de estados crece exponencialmente a medida que N aumenta, por lo que se ha notado un incremento en la cantidad de nodos expandidos por el algoritmo.
- ✓ Cada comunicación por petición de trabajo requerirá enviar mayor volumen de datos debido a que los tableros son más grandes.

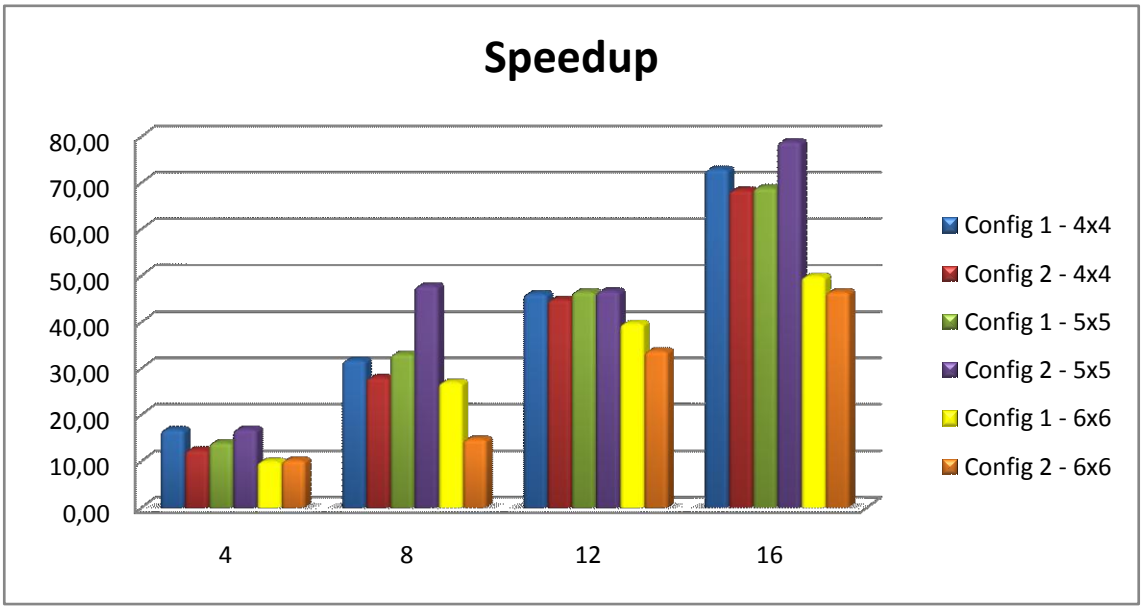

Figura 7.6: Gráfica del speedup de los mejores tiempos paralelos.

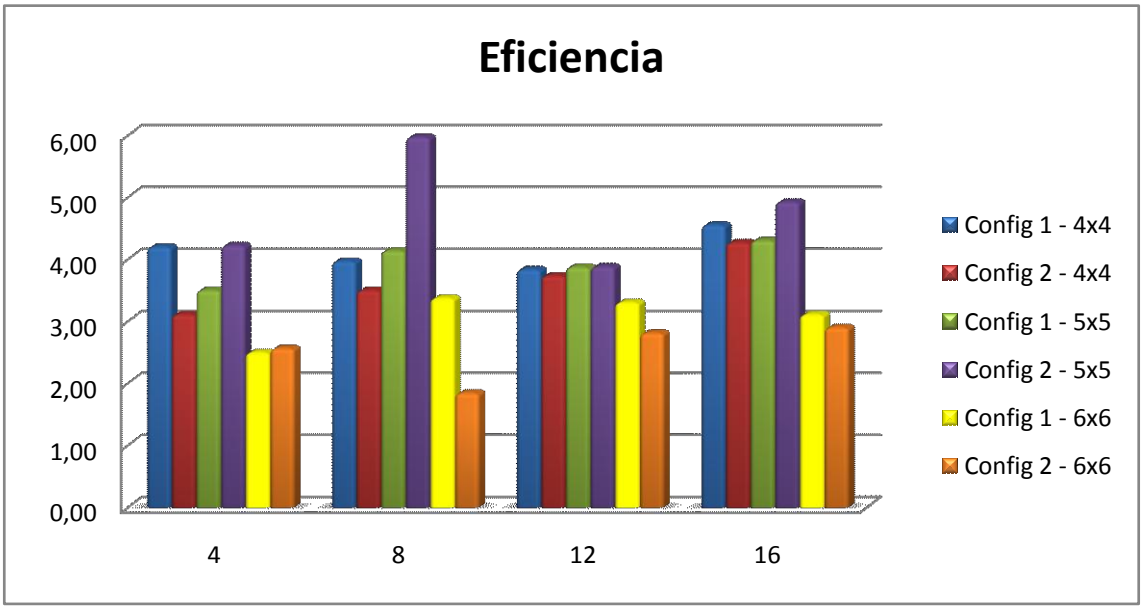

Figura 7.7: Gráfica de la eficiencia de los mejores tiempos paralelos.

8. Aplicación del problema del Puzzle a movimientos de robots

Las búsquedas, y en especial las búsquedas heurísticas, son un área de estudio de gran interés en Inteligencia Artificial. Un robot que desea ir desde un estado a otro puede ser visto como un agente. El estudio de agentes, y en particular agentes inteligentes, que intentan alcanzar un objetivo de forma de obtener el máximo rendimiento, es otra área de estudio de gran importancia en la IA por su aplicación a diversas problemáticas.

En esta sección se relacionan dichas ramas de estudio, y se plantea una generalización del problema del Puzzle, de forma de aplicarlo a problemas de robots con múltiples objetivos, y además se estudia una relación del problema con problemas de múltiples robots. Asimismo, se presenta una posible resolución del problema generalizado.

8.1 Planificación de movimientos

Un *problema de planificación* puede definirse de la siguiente manera: el mundo se encuentra en un determinado estado, pero se quiere que pase a otro estado distinto. El problema de planificación trata de cómo llegar desde un estado actual a través de una secuencia de movimientos a un estado meta. Estos problemas se resuelven aplicando algoritmos de búsquedas que encuentren una secuencia de movimientos que permitan a un agente alcanzar un objetivo a partir de un estado inicial. [WEE05]

En IA es muy común el término *agente*. Un agente es una entidad que percibe el ambiente a través de sensores, procesa las entradas y actúa sobre el ambiente a través de actuadores. Un agente puede ser un humano, un *robot*, o un software. En particular, un *agente inteligente* es aquel que actúa para maximizar el resultado esperado.

La estructura de un agente se compone de su arquitectura (sensores, motores, dispositivos de cómputo, etc) y de un programa que implementa su comportamiento, es decir toma las entradas percibidas, las procesa y retorna las acciones que deberán realizar los actuadores. [RUS03]

En la sección 3 de este trabajo, se discutieron las distintas técnicas de búsqueda en espacios de estados. Luego de plantear el problema y encontrar una solución, se ejecuta paso a paso las acciones, ignorando lo que se percibe ya que se asume que la solución encontrada siempre es exitosa. En la figura 8.1 se muestra el algoritmo básico para este tipo de agentes:

I ← Adquirir/leer el estado inicial

META? ← Elegir/leer mecanismo de detector la meta

Suc ← seleccionar/leer la estrategia de ramificación

Solución ← **Búsqueda** (I, META?, Suc)

REALIZAR (Solución)

Figura 8.1: Algoritmo de un agente simple solucionador de problemas

Estos algoritmos que encuentran planes óptimos son necesarios en el área de robótica para resolver tareas, ya que se requiere convertir especificaciones de tareas de alto nivel (por ejemplo: caminar de un punto a otro esquivando obstáculos) a descripciones de bajo nivel, que especifiquen qué movimientos realizar.

La planificación de movimientos de robot ignora las restricciones del robot y se basa en encontrar solo los movimientos que debe realizar el robot para alcanzar el objetivo, ya sea de forma óptima o no. La *planificación de trayectoria* toma la planificación realizada y determina cómo debe moverse el robot tomando en cuenta las limitaciones mecánicas del mismo. [LAV06]

Las tecnologías de robots humanoides, que realizan tareas que actualmente son ejecutas por un humano, han progresado rápidamente estos últimos años. Se han realizado grandes esfuerzos por desarrollar métodos prácticos de planificación para tareas tales como navegación, agarrar y manipular objetos, posicionamientos del cuerpo del robot, entre otros.

Las estrategias para planificación de caminos y esquivo de obstáculos han sido estudiados por diversos autores. En particular, los robots bípedos pueden sortear obstáculos pasándolos por encima. Una de las estrategias para planificar caminos para alcanzar una meta es discretizar las posibles posiciones del pie robótico, por ejemplo a N posiciones, para luego implementar un algoritmo similar al utilizado como base para el caso de estudio, y una heurística, de modo de minimizar el numero y complejidad de los pasos dados por el robot. [KUF03]

En el caso de la manipulación de objetos, los estados iniciales y finales son las posturas del cuerpo del robot. El movimiento de un objeto de un lugar a otro implica tres tareas: posicionar el robot para agarrar el objeto, luego de agarrarlo transferir el objeto a la posición final, y luego soltarlo, posicionar al robot en su posición de descanso. Se debe encontrar una secuencia de movimientos que conecte cada fase.

Como se ha visto, primero se puede calcular el camino, aplicando paralelismo para mejorar el tiempo de respuesta, y luego utilizar dicha solución para hacer mover al robot.

8.2 El problema del Puzzle y su relación con la robótica

Supongamos que se tiene un grafo G, y además un robot móvil en uno de los vértices (*s*), y otros vértices con obstáculos móviles. Los obstáculos y el robot se mueven a través de los vértices. Cada vértice puede contener a lo sumo una entidad (robot u obstáculo). En cada paso, el robot o uno de los obstáculos se mueve desde su posición (*v*) a una posición libre adyacente. El objetivo es mover el robot a un vértice (*t*) usando la menor cantidad de pasos posibles. Este problema es un *problema de planeamiento de movimientos en un grafo con un solo robot*.

Una generalización del problema a multirobots, sería el problema del Puzzle N^2-1, donde hay N^2-1 robots y no hay obstáculos, y cada robot debe trasladarse desde su posición inicial hacia la posición destino. [PAP94]

Otra forma de relacionar el problema sería pensar que la ficha "hueco" es un robot, que tiene capacidad para tomar y sostener un objeto, moverse a la posición que ocupaba éste, y soltar el objeto en la posición que ocupaba el robot anteriormente. Cada objeto puede ser visto como un obstáculo o un artículo que tiene un destino asignado. A partir de una configuración inicial, el robot debe ordenar los obstáculos/artículos según la configuración final elegida.

Estudiar esta clase de problemas es de interés, ya que se aplican a problemas de movimiento de robots en espacios geométricos, tal como se comentó en la sección anterior, moviendo obstáculos en caso de ser necesario.

8.3 Generalización del problema del Puzzle a múltiples objetivos

Con el fin de ampliar el problema clásico del Puzzle a múltiples objetivos, se ha creado un problema denominado *4-Puzzle N^2-1*, que consta de las siguientes características:

- ✓ El problema consiste de una sucesión de M tableros generados al azar que admiten 4 soluciones posibles, cada una de las cuales es un "tablero objetivo" T_i con 1 <= i <= 4, que se muestran en la figura 8.2 para N=5.

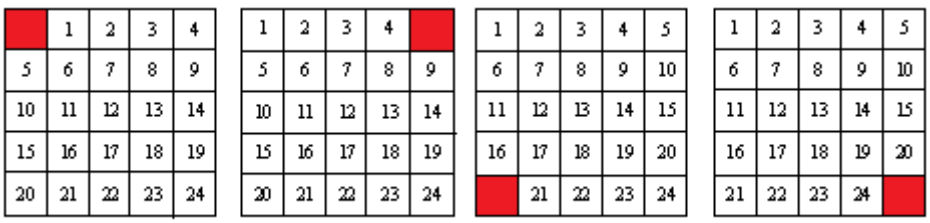

Figura 8.2: Tableros solución para el problema general.

- ✓ El análisis de los casos solubles y el espacio de soluciones posibles indicado en el punto 2.2.3 sigue siendo válido para cualquiera de los 4 "tableros objetivo" definidos. El análisis detallado de los "tableros objetivos" alcanzables es diferente para N par o impar, pero responde a lo expuesto en 2.2.3, lo que permite la distribución de tareas entre procesadores.

- ✓ Si se utiliza la métrica *H4* definida en la sección 7 como un estimador del objetivo más fácilmente alcanzable, y por como se generaron los tableros, estadísticamente para M creciente se balancea la distribución de los mismos entre los 4 T_i propuestos.

- ✓ La paralelización del problema se puede resolver en dos niveles: en principio una distribución funcional según el Ti objetivo y luego, la paralelización del algoritmo descripta en la sección 6.

- ✓ Interesa estudiar la resolución de este problema sobre clusters, determinando la bondad de la métrica *H4* como estimador del objetivo más probable de alcanzar en menor número de movimientos y también el paradigma de programación paralela a utilizar para maximizar speedup y eficiencia.

9. Conclusiones

Se ha presentado un análisis de la solución secuencial y paralela para el problema del Puzzle N^2-1 sobre clusters, incorporando en la implementación de diferentes heurísticas para la valuación de los nodos durante la búsqueda. Todas estas heurísticas se basan en la Distancia de Manhattan, pero cada una incorpora una mejora.

Se ha analizado la ventaja de utilizar una heurística mejorada respecto a la heurística clásica, tanto para el algoritmo secuencial como para el paralelo, y se ha estudiado speedup, eficiencia y superlinealidad para diferentes configuraciones de la arquitectura de cluster homogéneo y diferentes dimensiones y estados iniciales del problema.

El algoritmo paralelo, incorpora un parámetro LW, el cual indica la cantidad de nodos a procesar por iteración por cada procesador. Se observa que no existe un valor de LW óptimo para todas las pruebas, y se estudiaron las causas, llegando a la conclusión que dicho parámetro es muy dependiente del grafo que se genere durante la búsqueda y de cómo este espacio de estados es repartido entre los procesadores.

Se ha estudiado la aplicación del problema a casos de movimientos de robots, en particular multirobots y múltiples objetivos. Para lo último, se planteó una generalización del problema del Puzzle N^2-1.

10. Líneas de trabajo futuras

10.1 Resolución del problema del puzzle en un multicluster

Como se destacó en la sección 5, es importante realizar un análisis de la aplicación antes de realizar la migración de un problema a un multicluster. Las comunicaciones entre los clusters deben disminuirse, ya que son más costosas, más aun si están interconectados mediante una red no dedicada.

Para el problema del Puzzle, en cada cluster se podría tener un proceso dedicado a las comunicaciones entre clusters, de modo que los procesos dentro del cluster trabajen según la resolución estudiada, realizando el balance de carga localmente, y detección de comunicación localmente. Cada cierto tiempo, los gestores de comunicación de cada cluster deberían comunicarse, de modo de igualar la calidad de los nodos pendientes a procesar en cada cluster, detectar terminación, y comunicarse soluciones optimas encontradas hasta el momento.

10.2 Resolución del problema del puzzle en procesadores multicore

En la actualidad cada vez son más comunes los procesadores dual-core o quad-core entre los usuarios, que contienen dos y cuatro cores respectivamente.

Un procesador multicore combina dos o más cores independientes (CPU) en un único circuito integrado. Cada microprocesador multicore implementa multiprocesamiento dentro del mismo chip. Sus cores pueden compartir en algunos casos la memoria cache L2 (Intel Core 2) o pueden tener caches propias (AMD). En cualquier caso, los cores comparten la interconexión con el resto del sistema.

Las ventajas de este tipo de estos chips yace en la proximidad entre los cores, lo que posibilita que las señales entre los mismos atraviesen distancias más cortas, permitiendo trabajar más rápido, por ejemplo, a la circuitería de coherencia de cache.

A partir de unir varios chips, se puede crear una arquitectura NUMA de multicores, de modo que los cores dentro de un chip compartan la memoria principal, y puedan acceder remotamente a la memoria de otro chip, aunque ese acceso sea más costoso. También son comunes las arquitecturas donde los chips dual-core comparten la memoria principal, la cual es accedida a través de un bus compartido. La figura 10.1 muestra ambas arquitecturas.

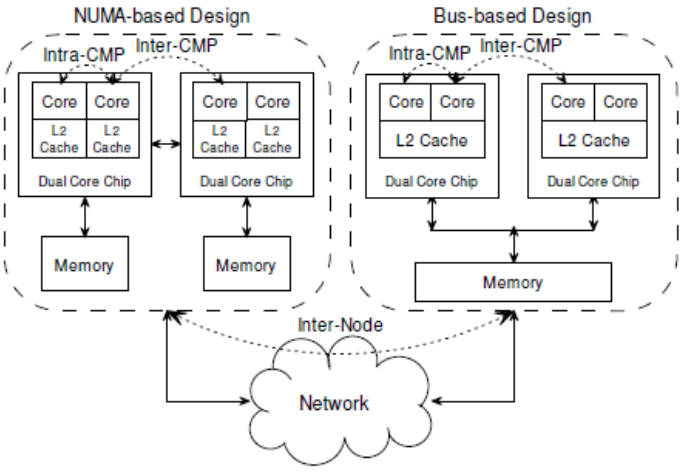

Figura 10.1: arquitecturas multicore (a) NUMA y (b) basadas en bus compartido

El paradigma de programación a utilizar en este tipo de arquitecturas es mediante memoria compartida. Los lenguajes de programación más difundidos en la actualidad son OPENMP y C utilizando Pthreads.

Las aplicaciones deberán replantearse para aprovechar este tipo de arquitecturas, incluyendo threads que se puedan asignar a distintos procesadores. Es de interés investigar la superlinealidad en las mismas, en especial para los problemas de optimización discreta. Los procesos trabajadores deberán aprovechar la jerarquía de memoria subyacente.

Por otro lado, como se comentó en la sección 5, los clusters se han impuesto en los últimos años como una opción eficiente en la relación costo/rendimiento. A partir de las arquitecturas multicores surgen los clusters de multicores. En estas arquitecturas hay tres niveles de comunicaciones: entre los cores en un mismo chip, entre chips pero dentro del mismo nodo, y la comunicación entre nodos vía red. El modelo de programación a utilizar en estos casos será un híbrido entre pasaje de mensajes y memoria compartida.

10.3 Resolución de la generalización del problema a múltiples objetivos en plataformas multicluster

Una alternativa de paralelización en plataformas multicluster del problema del Puzzle generalizado a múltiples objetivos, es tomar en cuenta la potencia de cálculo de los procesadores/clusters y resolver los problemas más simples en los clusters menos potentes y los más complejos en los de mayor potencia. En cada cluster la resolución puede realizarse utilizando la paralelización descripta en la sección de desarrollo.

A. Análisis de la formula de testeo de solubilidad

Como se ha visto en la sección 2, el espacio de estados para el problema del puzzle N^2-1 crece en orden factorial, siendo de tamaño $N^2!$, debido a que cualquier ordenamiento de las fichas puede ser tomado como un tablero válido. Sin embargo, el espacio de estados real es de tamaño $N^2!/2$, ya que su grafo tiene dos componentes conexas, de igual tamaño.

Se ha presentado en la sección 2.2.3 un algoritmo que, a partir de un tablero inicial y un tablero final, decidía si el tablero final podía ser alcanzado desde el tablero inicial. Este algoritmo utiliza una fórmula basada en contar el "número de inversiones" de las fichas de cada tablero, y luego compara la paridad de ambos resultados.

En este apéndice, se estudia informalmente el funcionamiento de la fórmula propuesta.

A.1 Configuraciones legales e ilegales

Teniendo en cuenta un estado final en particular, se puede decir que hay dos grupos de configuraciones: aquellas que se pueden resolver para ese estado final, y las que no tienen solución para dicho estado debido a que este es inalcanzable por estar en otra componente conexa.

Las primeras configuraciones, también llamadas configuraciones *legales*, se pueden obtener a partir del estado final, moviendo el hueco a sus posiciones adyacentes. Las otras configuraciones, a veces llamadas *ilegales*, se obtienen a partir de intercambiar las posiciones dos fichas vecinas de un tablero legal (que no sea el hueco).

A.2 Fórmula de solubilidad

Sea n_i la cantidad que denota el *número de inversiones* que tiene la ficha i, es decir la cantidad de piezas con menor número que aparecen después de ella, ya sea en la misma fila a su derecha, o en cualquier fila inferior, se realizan las siguientes proposiciones:

A.2.1 Proposición para N par

Si $NT = n_2 + n_3 + \dots n_{(N^2-1)} + x$, donde x es el número de fila del hueco, empezando a contar la primera fila como 1. (NT mod 2) se mantiene invariante luego de un movimiento legal.

Prueba:

Los movimientos horizontales del hueco no cambian el ordenamiento de las fichas, por lo que no agregan ni restan inversiones, es decir no modifican NT. Al contrario, los movimientos verticales si modifican NT.

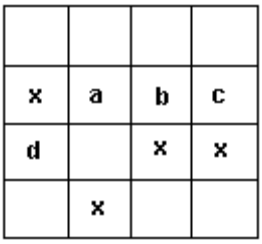

Figura A.1: Tablero con N par.

Teniendo en cuenta el tablero general de la figura A.1, supongamos se quiere mover el hueco hacia arriba. Ahora la fila del hueco es 1 menos que antes. La ficha "a" pasa a estar después de N-1 fichas (en este caso 3).

- ✓ Si las 3 fichas eran mayores que "a", ahora hay 3 inversiones más. Por lo tanto a NT se le resta 1 unidad, por el movimiento del hueco a una fila anterior, y suman 3 inversiones más, manteniendo la paridad de NT.

- ✓ Si había 2 fichas mayores que "a" y una menor, ahora hay dos inversiones más, y dado que "a" se posiciona después de la ficha que era menor, se resta una inversión a n_a. Por lo que a NT se le resta 1 (por el desplazamiento de la fila del hueco), se suman 2 inversiones (correspondientes a las inversiones que se agregan por las fichas mayores que ahora pasan a estar antes que "a"), y se le resta 1 inversión a n_a. NT se mantiene igual.

- ✓ Si había 3 fichas menores delante de "a", se restarían 3 inversiones a n_a, y 1 unidad por el desplazamiento del hueco. Manteniendo la paridad de NT.

- ✓ Se hace lo mismo para las demás combinaciones.

En general si T es el valor de la ficha que se mueve, supongamos que r fichas de las N-1 en cuestión son menores que T, y (N-1)-r son mayores. Sea V el número de inversiones antes del movimiento, y W el número de inversiones después del movimiento.

$$W = V – r + (N - 1 - r) -1 = V - 2r + N – 2$$

- ✓ Si V era par, se le agrega/resta un número par, dejando a W par.

- ✓ Si V era impar, se le agrega/resta un número par, dejando a W impar.

Por lo que un movimiento del hueco hacia arriba no cambia la paridad. Lo mismo se prueba para el movimiento del hueco hacia abajo, y estando la ficha que se mueve en cualquier columna.

A.2.2 Proposición para N impar

La proposición anterior no funciona para tableros con dimensión impar, ya que por cada movimiento vertical causa un salto entre una cantidad par de fichas, sumando a la cantidad previa de inversiones un número par de inversiones, por lo que preserva la paridad.

Si NT = $n_2 + n_3 + \ldots n_{(N^2-1)}$. (NT mod 2) se mantiene invariante luego de un movimiento legal.

Prueba:

x	a	b	c	d
e		x	x	x

Figura A.2: Tablero con N impar.

Teniendo en cuenta el ejemplo de la figura A.2. La prueba es similar a la anterior. Los movimientos verticales son los únicos que cambian el ordenamiento de las fichas, por lo que cambiarían NT.

- ✓ Supongamos las fichas b, c, d y e son mayores que "a". Al mover "a" hacia abajo se debe sumar a NT cuatro inversiones. NT mantiene la paridad.
- ✓ Supongamos que 3 fichas de {b,c,d,e} son mayores que "a", y 1 es menor. Dado que "a" pasa a estar delante de la ficha menor luego del movimiento, a NT se le resta una inversión. Dado que "a" pasa a estar delante de las 3 fichas mayores, se le suma tres inversiones a NT. Por lo anterior, NT mantiene la paridad.
- ✓ Supongamos que 2 fichas de {b,c,d,e} son mayores que a, y 2 son menores. NT se mantiene (ya que se restan 2 inversiones porque "a" pasa a estar delante de las dos fichas menores luego del movimiento, y se le suman dos inversiones, ya que "a" pasa a estar delante de las dos fichas mayores).

Haciendo una generalización similar a la anterior:

$$W = V - r + (N - 1 - r) = V - 2r + N - 1.$$

Como (N – 1) es par y 2r es par: si V era impar, luego del movimiento legal permanece impar, y si V era par entonces luego del movimiento permanece par.

A.3 Conclusiones

De lo estudiado anteriormente, se puede decir:

- ✓ Si el tablero final tiene NT par, luego de cada movimiento NT se mantiene par. Entonces, a partir de un tablero con NT par se puede alcanzar sólo tableros con NT par.
- ✓ Si el tablero final tiene NT impar, luego de cada movimiento NT se mantiene impar. Entonces, a partir de un tablero con NT impar se puede alcanzar sólo tableros con N impar.

Por lo tanto, el grafo asociado al Puzzle siempre es bipartito: tiene dos componentes conexas, una consiste en tableros con permutación par y la otra con tableros con permutación impar.

Referencias

[AND95] Anderson T., Culler D., Patterson D. A Case for NOW (Networks of Workstations). IEEE Micro 1995; 15(1): pp. 54-64.

[BAK99] Baker M., R. Buyya. "Cluster Computing at a Glance". R. Buyya Ed., High Performance Cluster Computing: Architectures and Systems, Vol. 1, Prentice-Hall, Upper Saddle River, NJ, USA, pp.3-47,1999.

[BAS99] Basney J., M. Livny. "Deploying a High Throughput Computing Cluster". R. Buyya Ed., High Performance Cluster Computing: Architectures and Systems, Vol. 1, Prentice-Hall, Upper Saddle River, NJ, USA, pp. 116-134, 1999.

[BOH02] Bohn C., Lamont G. Load Balancing for Heterogeneous Clusters of PCs. Future Generation Computer Systems, 2002; 18(3): 389-400.

[BUY99] Buyya R. High Performance Cluster Computing: Architectures and Systems. Prentice-Hall; 1999.

[COR90] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill; 1990.

[CUL98] Culberson, J., and J. Schaeffer. Pattern Databases, Computational Intelligence, Vol. 14, No. 4, 1998, pp. 318-334

[DIJ80] Dijkstra E., Scholten C. Termination detection for diffusing computations. Information Processing Letters 1980; 11(1):1-4.

[FER96] Ferreira A., Pardalos P. Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques. New York: Springer; 1996.

[FIT05] Fitch R., Butler Z., Rus D. Reconfiguration Planning Among Obstacles for Heterogeneous Self-Reconfiguring Robots. In: Proc. of the IEEE International Conference on Robotics and Automation; 2005. p. 117-124.

[GER05] Geraerts R., Overmars M.H. Reachability analysis of sampling based planners. In: IEEE International Conference on Robotics and Automation, pp. 406–412 (2005)

[GRA03] Grama A., Gupta A., Karypis G., Kumar V. An Introduction to Parallel Computing. Design and Analysis of Algorithms. Pearson Addison Wesley; 2003.

[GRA99] Grama A., Kumar V. State of the art in parallel search techniques for discrete optimisation problems. IEEE Trans. on Knowledge and Data Engineering, 1999.

[HAN92] Hanson O., Mayer A., Yung M. Criticizing Solutions to Relaxed Model Yields Powerful Admissible Heuristics. Information Sciences 1992; 63(3): 207-227.

[HEL90] Helmbold D., McDowell C. Modeling speedup(n) greater than n. IEEE Trans. on Parallel and Distributed Systems, 1990; 1(2): 250-256.

[HWA93] Hwang K. Advanced Computer Architecture. Parallelism, Scalability, Programmability. McGraw Hill; 1993.

[JOH79] Johnson W. W. and Storey W.E. Notes on the 15'-Puzzle. Amer. J. Math. 2(1879), 397-404.

[JOS03] Joseph J., Fellenstein C. "Grid Computing". On Demand Series. IBM Press (December 30, 2003).

[JUH04] Juhasz Z. (Editor), Kacsuk P. (Editor), Kranzlmuller D. (Editor). "Distributed and Parallel Systems: Cluster and Grid Computing". The International Series in Engineering and Computer Science. Springer; 1 edition (September 21, 2004)

[KOR05] Korf R. E., Schultze P. Large-scale parallel breadth-first search. In: Proc. of the 20th National Conference on Artificial Intelligence; Pittsburgh, USA; 2005. p. 1380-1385.

[KOR00] Korf, R. E. "Recent Progress in the Design and Analysis of Admissible Heuristic Functions". In: Proc. of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence. USA; 2000. p. 1165-1170.

[KOR96] Korf R. E., Taylor R. "Finding Optimal Solutions to the Twenty-Four Puzzle", 1996. In: Proc. of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)

[KUF03] Kuffner J., Nishiwaki K., Kagami S., Inaba M., Inoue H. Motion Planning for Humanoid Robots. In: Proc. 11th Int'l Symp. of Robotics Research (ISRR 2003).

[LAM04] Lambur H, Shaw B. Parallel State Space Searching Algorithms. 2004. www.metablake.com/parallel_search_project

[LAV06] LaValle S. Planning Algorithms. Cambridge University Press; 2006.

[LEO01] Leopold C. Parallel and distributed computing. A survey of models, paradigms, and approaches. New York: Wiley; 2001.

[MAN02] Manquinho V., Marques-Silva J. Search Pruning Techniques in SAT-Branch-and-Bound Algorithms for Binate Covering Problem. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2002; 21(5): 505-516.

[MAR90] Martello, S., Toth, P. Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc., New York (1990)

[OGU03] Ogura S, Nakada H, Matsuoka S. "Evaluation of the inter-cluster data transfer on Grid environment". Proceedings of CCGrid 2003 , pp. 374-381, May 2003.

[PAP94] Papadimitriou C., Raghavan P., Sudan M., Tamaki H. Motion Planning on a Graph. In: Proc. of the 35th Annual Symposium on Foundations of Computer Science; 1994.

[PAR95] Parberry I. A Real Time Algorithm for the (n2-1) Puzzle. Information Processing Letters 1995;56(1): 23-28.

[PEA94] Pearl, J. Heuristics. Addison-Wesley, Reading, MA, 1984.

[PFI98] Pfister, Gregory F. In search of clusters, the outgoing battle in lowly parallel computing. Prentice Hall, 1998.

[QUI93] Quiin M. J. Parallel Computing: Theory and Practice. McGraw-Hill Companies; 1993.

[RAO93] Rao V. N., Kumar V. On the efficiency of parallel backtracking. IEEE Trans. on Parallel and Distributed System, 1993; 4(4): 427-437.

[RAT86] Ratner D. and Warmuth W. Finding a shortest solution for the NxN extension of the 15-puzzle is intractable. AAAI-86, 168-172

[RAT90] Ratner D., Warmuth M. The (n2−1)-puzzle and related relocation problems. Journal for Symbolic Computation 1990; 10(2):11–137.

[REI93] Reinefeld A. Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*. In: Proc. of the 13th International Joint Conference on Artificial Intelligence; Chambery Savoi, France; 1993. p. 248-253.

[RUS03] Russel, S., Norvig, P. Artificial Intelligence, A modern Aproach. Pearson Prentice Hall; 2003.

[SAN07] Sanz V. Paralelización de N-Puzzle. Technical Report 2007.

[SAN07] Sanz V., Chichizola F., Naiouf M., De Giusti L.,De Giusti A. Superlinealidad sobre Clusters. Análisis experimental en el problema del Puzzle N2-1. In: Proc. of the

XIII Congreso Argentino de Ciencias de la Computación; Chaco-Corrientes, Argentina; 2007. p. 1300-1309.

[SER06] Sergienko I., Shylo V. Problems of discrete optimization: Challenges and main approaches to solve them. New York: Springer; 2006; 42(4):465-482.

[WEE05] De Weerdt M., Ter Mors A., Witteveen C. Multi-agent Planning, An introduction to planning and coordination. Dept. of Software Technology, Delft University of Technology; 2005.

[WIL05] Wilkinson B., Allien M. Parallel Programming: Techniques and Applications Using Network Workstation and Parallel Computers. Pearson Prentice Hall; 2005.