Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5438 - Inteligencia Artificial I
Trimestre Abril - Julio 2021
Prof. Carlos Infante
Amin Arriaga 16-10072, David Segura 13-11341, Wilfredo Graterol 15-10639

Búsqueda Informada: A* vs IDA*

July 25, 2021

1 Introducción

Las búsquedas informadas son algoritmos muy importantes en el área de inteligencia artificial, a pesar de que hoy en día la rama más famosa es la de Machine Learning. Entre estos algoritmos, A* e IDA* destacan bastante por su rendimiento en distintos problemas, y ambos comparten la carácterística de usar una función `h` llamada *heurística*, el cual toma un estado cualquiera del problema y estima el costo de llegar hasta algún estado objetivo. Estas funciones deben cumplir varias propiedades, como admisibilidad y consistencia, para que puedan ser útiles en la resolución de problemas. Mientras mejor sean estas heúristicas, más eficiente serán las funciones que lo usan. En particular, existen un tipo de heurística llamado Pattern DataBase (PDB) que reducen la complejidad del problema original al proyectarlo sobre un espacio de soluciones más pequeño y almacenan las soluciones de cada posible estado de este nuevo problema en una base de datos, se repite este proceso con distintas proyecciones y luego en el problema original se utilizan estos valores almacenados por estados para calcular una heurística. Si las proyecciones cumplen que al sumarse siguen cumpliendo las carácterísticas de admisibilidad y consistencia, se dicen que son PDBs aditivas, en caso contraria, se dice que no son aditivas y en general se toma el máximo valor entre todos ellos. En este informe estudiaremos el rendimiento de A* e IDA* con los distintos tipos de poda de duplicados, utilizando PDBs y sobre los problemas de N Puzzle, Torres de Hanoi, Top Spin y el Cubo de Rubik en distintas dificultades, así como sus árboles de búsqueda. Nos concentraremos en el tiempo de ejecución, memoria usada y número de estados por segundos generados para realizar el análisis.

2 Estructura del Repositorio

El repositorio presenta la siguiente estructura de archivos:

```
proyecto-1-ci5437/
|__ benchmarks/...
|__ bin/...
|__ generators
|   |__ HanoisTowersGenerator.py
|   |__ SlidingTileGenerator.py
|   |__ TopSpinGenerator.py
|__ informe.pdf -> src/L_informe/informe.pdf
|__ Makefile
|__ papers/...
|__ pdbs/...
|__ psvn/...
|__ puzzles/...
|__ README.md
|__ resources/...
|__ search_tree/...
|__ src
    |__ head.hpp
```

```
|__ heuristics.cpp
|__ heuristics.hpp
|__ InformedSearchs.cpp
|__ InformedSearchs.hpp
|__ L_informe/...
|__ main.cpp
|__ Node.cpp
|__ Node.hpp
|__ NodesPriorityQueue.cpp
|__ NodesPriorityQueue.hpp
|__ PriorityQueue.hpp
```

donde

- `benchmarks/` contiene los casos de prueba para los distintos puzzles separados en directorios, así como los archivos necesarios para generar nuevos casos de prueba.
- `bin/` contiene los archivos binarios que resuelven algún puzzle. Los archivos en este directorio tienen el formato `P.out` donde `P` es el nombre de algún puzzle. Estos archivos no son agregados al repositorio.
- `generators/` contiene los generadores de archivos `.psvn`. Los que se encuentran actualmente son:
  - `generators/HanoisTowersGenerator.py` tal que al ejecutarse con `P > 2` y `D > 1`, imprime un PSVN el puzzle de las Torres de Hanoi con `P` astas y `D` discos.
  - `generators/SlidingTileGenerator.py` tal que al ejecutarse con `M > 0` y `N > 0`, imprime un PSVN el puzzle de Sliding Tiles con dimensión `M x N`.
  - `generators/TopSpinGenerator.py` tal que al ejecutarse con `K > 1` y `N > K`, imprime un PSVN el puzzle Top Spin con `N` tokens y un 'turntable' de longitud `K`.
- `pdbs/` contiene los archivos necesarios para generar PDBs para los distintos puzzles a estudiar. Revise el archivo `pdbs/README.md` para más información.
- `psvn/` contiene el código fuente para compilar la API de PSVN.
- `puzzles/` contiene archivos `.psvn`.
- `search_tree` contiene los archivos necesarios para realizar el estudio de los árboles de búsqueda de cada puzzle.
- `src/` contiene el código fuente principal para compilar y ejecutar los distintos algoritmos de búsqueda informada que estudiaremos en este proyecto. Daremos una brvee explicación de cada archivo:
  - `src/Node.*` tiene la implementación de nodo que hemos usado durante las clases. También almacena la profundidad del camino parcial hasta ese nodo.
  - `src/PriorityQueue.hpp` tiene la implementación de una cola de prioridad genérica. Permite definir el tipo de dato que servirá para realizar las comparaciones, el tipo de los elementos que almacenará y la función de comparación. No se separó en archivos `.cpp` y `.hpp` debido a los problemas de C++ con los templates.
  - `src/NodesPriorityQueue.*` tiene otra implementación de una cola de prioridad pero basada en nodos, y que además de los métodos `empty`, `add` y `pop`, también tiene los métodos `find` que busca un nodo según el estado que almacena; y `replace_if_less` que, dado un nodo, verifica si la cola tiene otro nodo que representa al mismo estado que además tiene un costo parcial superior al nodo parámetro, entonces es sustituido por el nodo parámetro. Estas 2 funciones en una cola de prioridad común serían $O(n)$, lo cual no es deseable en las funciones de búsqueda.
  - `src/InformedSearchs.*` tiene las implementaciones de los algoritmos de búsqueda que se estudian en el proyecto. En particular, se encuentran los siguientes algoritmos:

    - A* sobre grafos.
    - A* sobre árboles.
    - A* con eliminación tardía de nodos.
    - IDA* sobre grafos.
    - IDA* sobre árboles.
    - IDA* con eliminación parcial de nodos.
- `src/heuristics.*` tiene las implementaciones de las distintas heúristicas que se usarán, principalmente de PDBs.
- `src/main.cpp` y `src/head.hpp` el cual, al compilarse y ejecutarse, te permite ingresar un estado inicial, así como escoger un algoritmo de búsqueda y una heurística de los implementados para resolver el puzzle. Consulte el archivo `Makefile` para más información.
- `src/L_informe/` contiene los archivos fuentes de latex necesarios para generar este informe.

3 Árboles de Búsqueda

En las *Figura 1 y 2* podemos notar que hay una inmensa diferencia entre la cantidad de nodos expandidos por cada puzzle con y sin poda de duplicados, lo cual también se refleja en el factor de ramificación mostrado en la *Tabla 1*, que disminuye considerablemente al aplicar la poda. Esto nos comienza a indicar la necesidad de aplicar alguna poda en los algoritmos de búsqueda. El ejemplo más claro de esto fue el puzzle de las Torres de Hanoi con 12 Discos, donde, sin poda de duplicados, ya expandía alrededor de 10^7 nodos a profundidad de 10, mientras que con poda se lograron expandir todos los posibles estados, llegando hasta una profundidad de 81 (lo cual nos muestra que podemos almacenar todo el puzzle en un solo pdb).

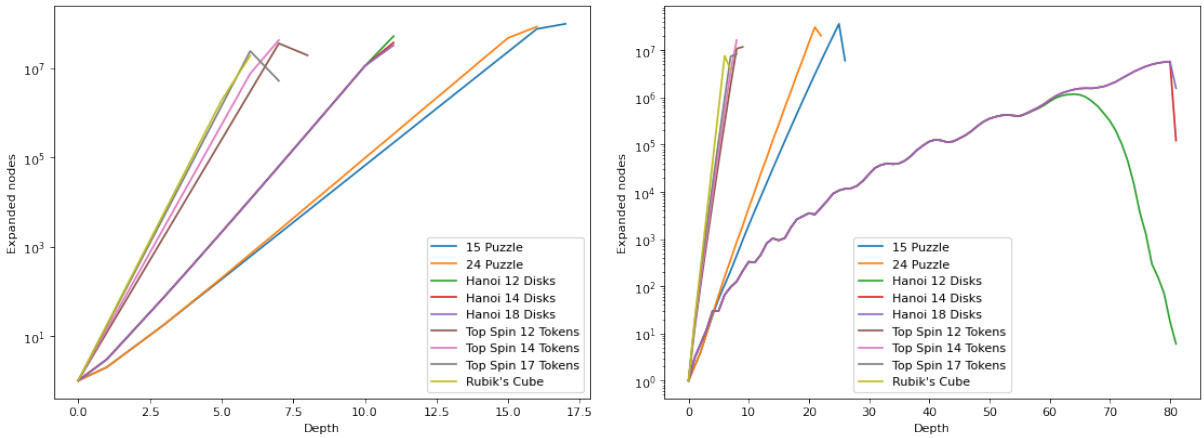
También se resalta la diferencia en la cantidad de nodos generados por profundidad entre cada puzzle. Tanto los Top Spins como el Cubo de Rubik tienen un crecimiento muy acelerado, alcanzando 10^7 estados nada más a profundidad de 10 con poda de duplicados, mientras que las Torres de Hanoi son los que presentan el crecimiento más lento. Esto no es necesariamente un indicador de dificultad del puzzle, pues alguno podría crecer lento y llegar a una gran profundidad, mientras que otro crezca rápido pero llegar hasta una profundidad baja. Esta medida de crecimiento tal vez es importante en el rendimiento de los distintos algoritmos de búsquedas.

Otro fenómeno interesante que podemos notar en las gráficas es que, para las Torres de Hanoi, los nodos expandidos hasta una profundidad de 60 son iguales para 12, 14 y 18 discos, lo cual es lógico, pues partiendo del estado final es imposible mover las piezas más grandes, así que los primeros movimientos son exactamente iguales. Eventualmente hasta cierta profundidad las Torres de Hanoi con 14 y 18 discos tambíen serían exactamente iguales y luego el de 14 comenzaría a disminuir la cantidad de nodos expandidos.

Por último, la cantidad de estados generados por segundo también cambia drásticamente entre cada puzzle. Se generan alrededor de 7 veces más estados de 15 Puzzle que del cubo de Rubik. Esto se puede deber a varios factores, desde longitud en la representación de los estados de cada puzzle, así como la complejidad en las reglas de transición, donde, por ejemplo, las del cubo de Rubik son más difícles de aplicar que la de los N Puzzle. Sin embargo, no tenemos suficiente evidencia para determinar que factores afectan la cantidad de estados generados por segundo.

4 Casos de Prueba

Para cada puzzle se correrán los distintos algoritmos sobre 5 casos de pruebas con una dificultad lo suficientemente baja como para que no tarden más de 2 minutos y se tomará el rendimiento promedio de los algoritmos en cada uno de esos casos. Luego se correrá un caso de prueba lo más difícil posible, y se graficarán los nodos expandidos por cada profundidad. Decidimos usar esta metodología para no durar una gran cantidad de tiempo en pruebas díficiles y aún así obtener datos de distintos casos para evitar que, por mala suerte, usemos uno anormal, dándonos datos que no son comunes y lo tomemos como tal.



Figuras 1 y 2. A la izquierda, los nodos expandidos por profundidad en cada puzzle sin poda de duplicados. A la derecha, con poda de duplicados

	WITHOUT PRUNING		WITH PRUNING	
PUZZLE	GENERATED STATES	BRANCH FACTOR	GENERATED STATES	BRANCH FACTOR
15 Puzzle	208022078	3.23607	85067782	1.82282
24 Puzzle	152070607	3.45299	77296487	2.1758
Hanoi 12 Disks	65574199	5.68374	16777216	1
Hanoi 14 Disks	50932443	5.68308	64148653	1.08567
Hanoi 18 Disks	46133774	5.67978	65852152	1.08861
Top Spin 12 Tokens	58484323	12	24681413	4.17437
Top Spin 14 Tokens	50635066	14	20485330	6.7183
Top Spin 17 Tokens	30838399	17	16773307	8.0007
Rubik's Cube	21010255	18	12062869	13.2302

Tabla 1. Número de estados generados y factor de ramificación promedio de cada puzzle con y sin poda de duplicados.

4.1 15 Puzzle

Para el 15 Puzzle usamos como heurística la distancia Manhattan. Los 5 casos de pruebas fáciles y el difícil escogidos junto a la longitud de sus soluciones fueron:

```
3 14 9 11 5 4 8 2 13 12 6 7 10 1 15 B     (46)
14 1 9 6 4 8 12 5 7 2 3 B 10 11 13 15     (45)
7 11 8 3 14 B 6 15 1 4 13 9 5 12 2 10     (46)
12 15 2 6 1 14 4 8 5 3 7 B 10 13 9 11     (47)
12 8 15 13 1 B 5 4 6 3 2 11 9 7 14 10     (50)
5 9 13 14 6 3 7 12 10 8 4 B 15 2 11 1     (57)
```

Podemos notar en las *Tablas 2 y 3* que para los casos fáciles IDA* fue mucho más eficiente que A*, siendo el mejor IDA* con poda de duplicados, y el peor A* con poda de duplicados. Sin embargo, para el caso difícil, A* con poda tardía de duplicados se vuelve el dominante, probablemente debido al aumento en la profundidad, lo cual se vuelve inconveniente para IDA*. Sin embargo, el uso de la memoria para A* fue considerablemente grande, alrededor de 4 GB, mientras que el de IDA* es casi despreciable como era de esperar. También podemos confirmar gracias a la *Figura 3* que el número de nodos expandidos por IDA* es mucho mayor que el de A* (note que la gráfica está en escala logarítmica), pero que el número de nodos expandidos por segundos es menor en A*.

Algo interesante que se ve en la gráfica es que el numero de nodos expandidos por IDA* fue exactamente el mismo con poda y poda parcial de duplicados. Suponemos que es porque para llegar a un estado

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	13.43522	0.20072	58182.431782
A* with Late Pruning	12.47072	0.1828	88860.62871
IDA* with Pruning	5.66057	0.00074	405317.264856
IDA* with Partial Pruning	10.7614	0.00082	216139.60118

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	633.547779	4.1973	25436.99234
A* with Late Pruning	305.977159	3.92621	77683.54042
IDA* with Pruning	349.547181	0.00082	404336.85832
IDA* with Partial Pruning	591.625852	0.00195	238931.14461

Tablas 2 y 3. A la izquierda promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de pruebas fáciles del 15 Puzzle por cada algoritmo de búsqueda. A la derecha tiempo, memoria y número de nodos expandidos en el caso de prueba difícil del 15 Puzzle para cada algoritmo de búsqueda.

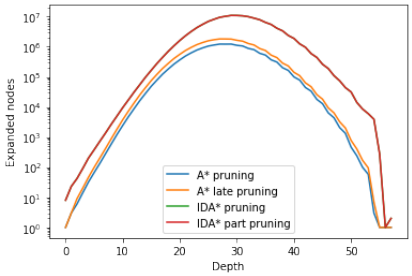


Figura 3. Número de nodos expandidos por profundidad en el caso de prueba difícil del 15 puzzle por cada algoritmo de búsqueda.

repetido puede ser usando el movimiento opuesto al último movimiento o a través de una gran cantidad de pasos que generen un ciclo, pues no es tan simple como moverse en un círculo. Para el primer caso, la poda parcial es suficiente para evitarlo. Mientras que para el segundo, lo mas probable es que el mismo algoritmo no permita realizar tal circuito, pues requeriría una gran cantidad de movimientos que no ayudan a llegar a una solución, y por lo tanto alcanzaría un costo que impediría seguir expandiendo esa rama. Así, al sólo ser necesario podar los estados padres y como A* con poda tardía de duplicados genera más nodos por segundo que con poda de duplicados, se obtiene que A* con poda tardía es mucho más eficiente que la poda total.

4.2 24 Puzzle

Para el 24 Puzzle usamos como heurística PDBs aditivos, particionando el puzzle tal como se muestra en la *Figura 4*. Lo 5 casos de pruebas fáciles y el difícil escogidos junto a la longitud de sus soluciones fueron:

```
5 1 B 2 4 6 7 8 3 9 10 16 12 13 14 15 22 11 18 19 20 17 21 23 24 (16)
1 6 7 2 3 5 12 8 B 4 10 17 11 13 9 15 16 18 23 14 20 21 22 24 19 (20)
5 1 2 3 4 6 7 12 8 9 15 10 11 13 14 21 20 18 B 24 16 22 17 19 23 (20)
1 2 3 4 9 11 5 10 8 14 6 7 B 12 13 15 16 17 18 19 20 21 22 23 24 (20)
5 2 7 3 4 10 6 1 8 9 15 11 12 13 14 16 22 21 19 24 20 17 23 18 B (20)
11 5 7 2 4 B 1 3 8 9 6 10 18 16 14 12 20 13 23 19 17 21 15 24 22 (53)
```

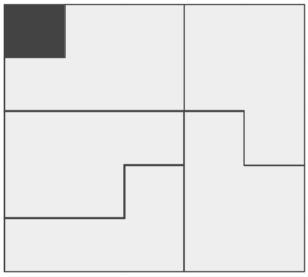
Utilizamos estos casos de prueba con poca dificultad pues, a pesar de que las máquinas virtuales de *Google Colab* ofrecen una gran cantidad de memoria RAM, su procesador es muy lento, por lo que crear casos de prueba que tuvieran tiempos de ejecución más aceptable, como 2 a 5 minutos para los casos fáciles, es difícil, pues al no saber de antemano la longitud de la solución, tenemos que probar hasta que alcance una o decidamos detener la ejecución.

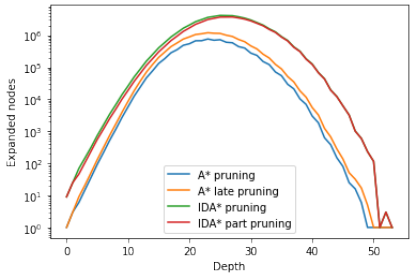
Notamos en las *Tablas 4 y 5* que, al igual que con el 15 Puzzle, IDA* con poda de duplicados es la más eficiente para los casos de prueba fáciles, pero IDA* con eliminación parcial de duplicados fue el que tardó más. Además, A* con poda total tuvo el mejor rendimiento para el caso difícil, diferiendo de 15 Puzzle en el que fue A* con poda tardía. Aún así, se mantiene el patrón de que IDA* empeora su rendimiento para los casos difíciles, probablemente por el aumento en la profundidad de la búsqueda.

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	11.21357	2.09321	7248.22198
A* with Late Pruning	15.43712	2.04771	8321.8615
IDA* with Pruning	10.94073	2.02139	35645.94902
IDA* with Partial Pruning	21.40956	2.02114	18225.15394

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	1416.934345	5.11623	6111.89222
A* with Late Pruning	1749.893619	4.74151	7963.92183
IDA* with Pruning	1600.302062	2.02169|	33210.60584
IDA* with Partial Pruning	2818.825964	2.02227	16736.42204

Tablas 4 y 5. A la izquierda promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de pruebas fáciles del 24 Puzzle por cada algoritmo de búsqueda. A la derecha tiempo, memoria y número de nodos expandidos en el caso de prueba difícil del 24 Puzzle para cada algoritmo de búsqueda.





Figuras 4 y 5. A la izquierda, partición del 24 Puzzle que usamos para generar los Pattern Data Bases. A la derecha, Número de nodos expandidos por profundidad en el caso de prueba difícil del 24 puzzle por cada algoritmo de búsqueda.

La *Figura 5* es muy parecida a la del 15 Puzzle, incluso presenta también un pico al final para IDA*, mientras que A* termina con una zona plana, indicando que para este punto solo expando los nodos necesarios para alcanzar la solución.

4.3 Towers of Hanoi 4 Pegs - 12 Disks

Tal como mencionamos en la sección 3, pudimos confirmar que todo el puzzle de las Torres de Hanoi con 12 Discos cabe en un solo PDB, ocupando un total de 1.6 GB aproximadamente. Debido a esto, probamos el caso de prueba más difícil que existe en las Torres de Hanoi, el cual es colocar todos los discos en un asta distinto del objetivo, cuya solución es de 81 pasos. Los otros casos de prueba con menor dificultad junto a la longitud de sus soluciones fueron[1]:

```
2 3 4 1 1 1 1 1 1 1 1 1   (3)
4 2 1 1 1 1 1 1 1 1 1 1   (2)
3 1 2 1 1 1 1 1 1 1 1 1   (4)
1 2 2 1 1 3 4 4 2 1 1 1   (29)
3 3 1 3 3 2 1 1 1 4 3 4   (61)
```

Como se muestra en las *Tablas 6 y 7*, los tiempos de ejecución son muy pequeños, a pesar de usar el caso más difícil de este puzzle. Esto debido a que los algoritmos sólo deben seguir la heurística, que almacena la distancia exacta hasta el objetivo desde cada estado, por lo que se expanden muy pocos nodos, tal como podemos apreciar en la *Figura 7*, llegando a máximo cerca de 100 nodos por profundidad.

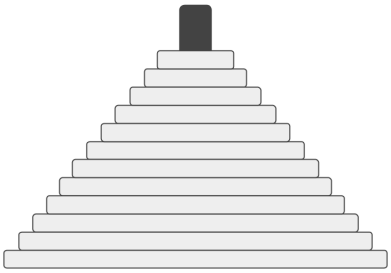
También podemos apreciar en la *Figura 7* que IDA* expandió muchos menos nodos que A*, lo cual es anti-intuitivo. No estamos seguros de por qué pasa eso, sin embargo, teorizamos que se debe al comportamiento de DFS que tiene IDA*, el cual, como sólo realiza una iteración ya que la heurística sabe exáctamente el costo desde el estado inicial, entonces va recorriendo un solo camino expandiendo pocos nodos pues la mayoría terminan siendo podados por la heurística. Mientras que A* debe expandir todos los nodos que compartan el mismo costo debido a su comportamiento similar a BFS.

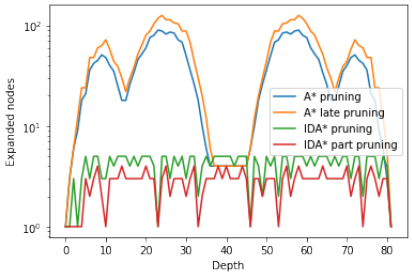
[1]Esta no fué la codificación real usada en PSVN, aquí hicimos una traducción para que sea más fácil de ver. La codificación real es binaria donde $1 \rightarrow 1000$, $2 \rightarrow 0100$, $3 \rightarrow 0010$ y $4 \rightarrow 0001$.

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.11585	1.64666	26344.41088
A* with Late Pruning	0.237714	1.6307	17369.61222
IDA* with Pruning	0.002496	1.62877	135817.30769
IDA* with Partial Pruning	0.003058	1.62949	74231.52387

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.00304	1.62904	17839.81965
A* with Late Pruning	0.00594	1.62584	15229.21397
IDA* with Pruning	0.00069	1.62584	88464.49656
IDA* with Partial Pruning	0.00083	1.62876	61280.32448

Tablas 6 y 7. A la izquierda, tiempo, memoria y número de nodos expandidos en el caso de prueba difícil de las Torres de Hanoi con 12 Discos para cada algoritmo de búsqueda. A la derecha, promedio del tiempo, memoria y número de nodos expandidos entre los casos fáciles de las Torres de Hanoi con 12 Discos para cada algoritmo de búsqueda.





Figuras 6 y 7. A la izquierda, representación de los PDBs que usamos para las Torres de Hanoi con 12 Discos. Básicamente usamos un solo PDB que almacena todo el puzzle. A la derecha, Número de nodos expandidos por profundidad en el caso de prueba difícil de las Torres de Hanoi con 12 Discos por cada algoritmo de búsqueda.

Otro fenómeno extraño es que IDA* con poda parcial de duplicados expandió menos nodos que con poda de duplicados. Para esto si no tenemos alguna posible explicación pues no sabemos como funciona la poda parcial de PSVN. Nuestra poda total consiste en almacenar el camino actual usando tablas de hash y verificar si el nodo hijo se encuentra en dicho camino, de ser así, no se expande.

4.4 Towers of Hanoi 4 Pegs - 14 Disks

Para las Torres de Hanoi con 14 Discos usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 8*. Para este puzzle decidimos usar un caso difícil y otro medio difícil, pues IDA* es muy poco eficiente en comparación a A* para resolver este problema, entonces en el difícil probamos únicamente A* para sacarle todo su potencial, y en el caso medio-difícil estudiamos tanto A* como IDA*.

```
4 4 4 4 4 4 4 4 4 2 2 2 2 2   (57)
4 4 4 4 4 4 4 4 4 4 4 4 4 4   (113)
```

Lo mas importante que se puede notar en la *Tabla 8* es la diferencia en eficiencia entre A* e IDA* para este puzzle, tardando el primero menos de un segundo para resolver el caso medio-difícil, mientras que IDA* tardó cerca de una hora. La razón de esto la podemos observar en la *Figura 9*, donde el número de estados expandidos por IDA* es del orden de 10^8, mientas que A* apenas llega a 10^3. Sin embargo, a partir de una profundidad de 20, el número de nodos expandidos por IDA* decrece abruptamente, y se mantiene en el orden de los 10 nodos hasta llegar a la solución. Creemos que la razón de esto es que para el paso número 20 los dos bloques más pesados ya se encuentran en la posición correcta, por lo que el primer bloque de la heurística, el que se encuentra a la izquierda de la *Figura 8*, pasa a convertirse en el problema entero, llegando a una caso muy parecido al de las Torres de Hanoi con 12 discos. Lo que apoya nuestra teoría es que desde la profundidad 20, la *Figura 9* es muy parecida a la *Figura 7*.

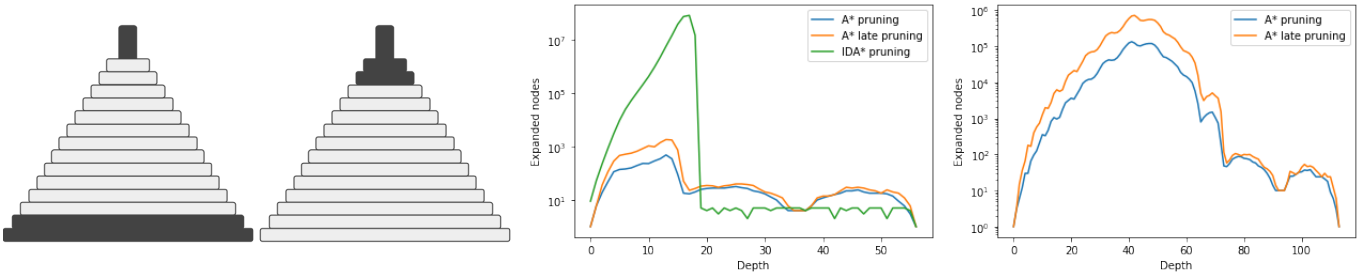
Otra cosa por destacar es que la ineficiencia de IDA* en este puzzle coincide con el hecho de que las Torres de Hanoi es el puzzle con crecimiento más lento y longitud de soluciones más largo que hemos analizado.

Por otro lado, A* con poda total fue más eficiente que A* con poda tardía, tal como se muestra en la *Tabla 9*. Siendo por lo tanto, el más eficiente en este algoritmo. Además, aunque no lo mostramos aquí,

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.206307	3.76336	17265.53147
A* with Late Pruning	0.616672	3.76187	20036.58347
IDA* with Pruning	3478.389176	1.72873	69633.77349

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	122.425431	5.11321	18364.37072
A* with Late Pruning	404.216894	5.0844	27497.23271

Tablas 8 y 9. A la izquierda y derecha tiempo, memoria y número de nodos expandidos en el caso de prueba medio-difícil y difícil respectivamente de las Torres de Hanoi con 14 Discos.



Figuras 8, 9 y 10. A la izquierda, representación de los PDBs usados para la heurística de las Torres de Hanoi con 14 Discos. En el medio y a la derecha, número de nodos expandidos por profundidad en los casos de prueba medio-difícil y difícil respectivamente de las Torres de Hanoi con 14 Discos.

para los casos de prueba fáciles, todos los algoritmos de búsqueda tardan prácticamente nada, pero el algún cambio en algúno de los dos bloques más pesados hace que IDA* pase una gran cantidad de tiempo analizandolo. Es por eso que decidimos que con estos 2 casos de prueba eran suficientes para realizar nuestro análisis.

4.5 Towers of Hanoi 4 Pegs - 18 Disks

Para las Torres de Hanoi con 18 Discos usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 11*. Para este puzzle el caso de prueba difícil solo fue probado sobre A*, pues IDA* tardó casi una hora en un caso de las Torres de Hanoi con 14 Discos cuando A* tardó menos de un segundo. Por lo que para este puzzle podría tardar una cantidad exagerada de tiempo. Lo 5 casos de pruebas fáciles y el difícil escogidos junto a la longitud de sus soluciones fueron

```
3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   (1)
4 3 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1   (4)
2 4 3 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1   (6)
2 2 1 2 1 4 4 1 1 1 1 1 1 1 1 1 1 1   (18)
1 4 4 4 4 3 4 4 1 1 1 1 1 1 1 1 1 1   (29)
1 2 2 1 2 4 4 2 3 2 4 1 4 1 1 1 1 1   (77)
```

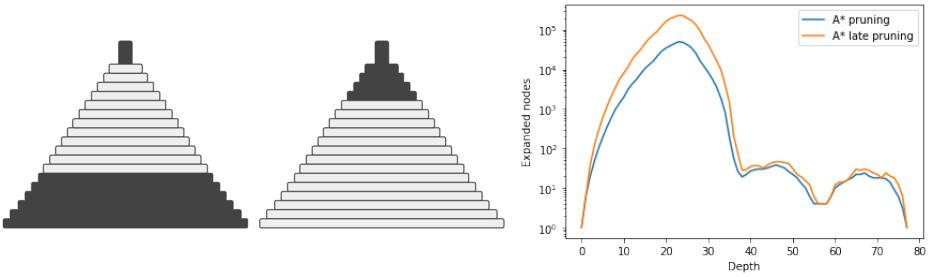
Para este puzzle ocurre práticamente lo mismo que con las Torres de Hanoi con 14 discos, si los 6 discos más pesados están en la posición correcta, entonces la primera heurística guiará al algoritmo directo a la solución sin hacerle expandir muchos nodos, tal como se ve en los tiempos de ejecución de la *Tabla 10*, los cuales son muy bajos. Sin embargo, mover un solo disco de los 6 más pesados hace que la búsqueda se vuelva bastante extensa. En la *Tabla 11* se nota como A* pasa de tardar menos de 1 segundo a 34 segundos moviendo un solo bloque. Considerando el rendimiento de IDA* en las Torres de Hanoi con 14 discos, no lo probamos con casos difíciles para este puzzle pues el tiempo de busqueda se vuelve demasiado largo.

Al igual que en el puzzle anterior, A* obtuvo el mejor rendimiento. Algo interesante de las grásficas de A* en las Torres de Hanoi, es que todas presentan un comportamiento ondulatorio, a diferencia de los N Puzzle que eran cuadráticos invertidos. Esto puede deberse a que hay etapas en el puzzle en las que la heurística logra guiar mejor a los algoritmo.

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.00247	5.46583	7802.20782
A* with Late Pruning	0.0039	4.76527	5996.77999
IDA* with Pruning	0.00097	4.76543	37919.41305
IDA* with Partial Pruning	0.00132	4.76522	20988.13621

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	34.306718	9.67172	14233.83024
A* with Late Pruning	132.832168	7.47716	17372.49369

Tablas 10 y 11. A la izquierda promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de pruebas fáciles de las Torres de Hanoi con 18 Discos por cada algoritmo de búsqueda. A la derecha memoria y número de nodos expandidos en el caso de prueba difícil de las Torres de Hanoi con 18 Discos.



Figuras 11 y 12. A la izquierda, representación de los PDBs usados para la heurística de las Torres de Hanoi con 18 Discos. A la derecha, número de nodos expandidos por profundidad en el caso de prueba difícil de las Torres de Hanoi con 18 Discos usando A.*

4.6 Top Spin 12 Tokens - Turntable of length 4

Para el Top Spin con 12 Tokens usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 13*. Los 5 casos de pruebas (todos difíciles) escogidos junto a la longitud de sus soluciones fueron

```
4 11 5 3 8 6 7 10 1 2 9 12   (9)
11 1 6 5 2 10 9 8 4 7 3 12   (7)
3 4 8 11 7 1 6 10 2 9 5 12   (8)
2 6 7 5 11 1 10 9 8 3 4 12   (9)
10 8 1 11 5 7 6 9 3 4 2 12   (10)
```

No hay mucho que decir en este puzzle ya que, al poder almacenar todo el puzzle en un solo PDB, entramos en el mismo caso que las Torres de Hanoi con 12 Discos. La heurística guiará a los algoritmos directamente a la solución sin mucha desviación, haciendo que el número de nodos expandidos en total sea bastante bajo independientemente de la dificulta del caso de prueba, por eso escogimos todos esos casos con la mayor dificultad posible. Los resultados se muestran en las *Tablas 12 y 13*. Lo único importante a mencionar es que, a diferencia de las Torres de Hanoi con 12 discos, en este caso el número de nodos expandidos por A* fue menor que IDA*.

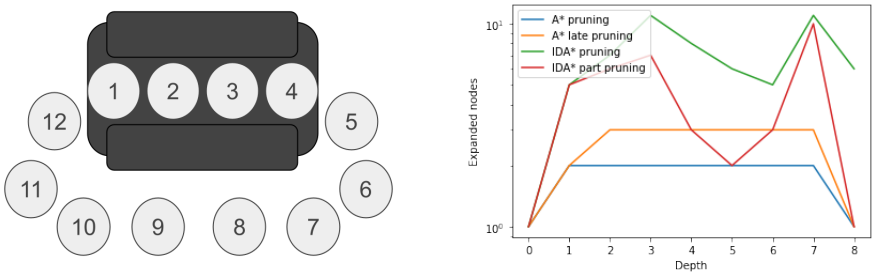
4.7 Top Spin 14 Tokens - Turntable of length 4

Para el Top Spin con 14 Tokens usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 13*. Los 5 casos de pruebas (todos difíciles) escogidos junto a la longitud de sus soluciones fueron

```
8 5 13 12 1 4 9 10 3 7 2 11 6 14  (11)
4 7 13 3 9 2 8 6 10 1 5 12 11 14  (10)
3 2 8 7 5 1 12 6 9 10 13 11 4 14  (11)
11 12 4 3 6 10 9 8 13 7 1 5 2 14  (9)
1 12 10 13 9 8 5 2 3 6 4 11 7 14  (11)
```

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.00301	1.32021	14545.14826
A* with Late Pruning	0.00343	1.00353	17762.08622
IDA* with Pruning	0.00025	1.00317	215466.56686
IDA* with Partial Pruning	0.00029	1.00382	129117.45619

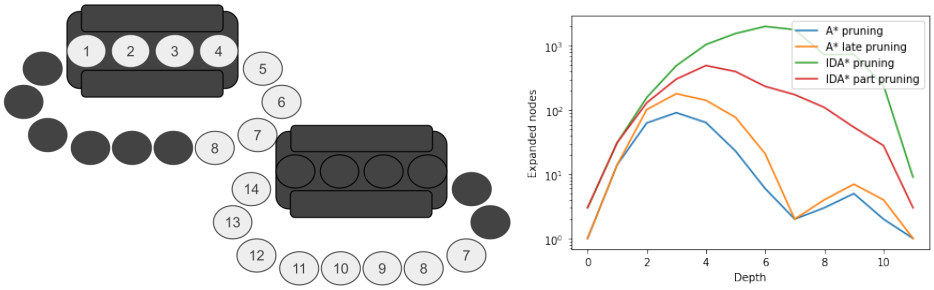
Tablas 12. Promedio en tiempo, memoria y número de nodos expandidos por segundo entre los 5 caso de pruebas difíciles de Top Spin con 12 Tokens para cada algoritmo de busqueda.



Figuras 13 y 14. A la izquierda, representación de los PDBs que usamos para Top Spin con 12 Tokens. Básicamente usamos un solo PDB que almacena todo el puzzle. A la derecha, Número de nodos expandidos por profundidad en el caso de prueba difícil de Top Spin con 12 Tokens por cada algoritmo de búsqueda.

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.04982	3.18009	6906.86635
A* with Late Pruning	0.06393	2.82195	8830.16799
IDA* with Pruning	0.02773	2.82087	148104.28168
IDA* with Partial Pruning	0.0162	2.82137	78291.78362

Tabla 13. Promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de prueba fáciles de Top Spin con 14 Tokens por cada algoritmo de búsqueda.



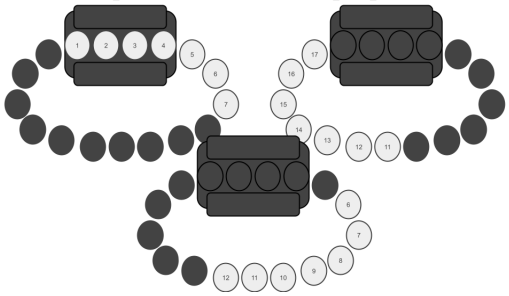
Figuras 15 y 16. A la izquierda, representación de los PDBs que usamos para Top Spin con 14 Tokens. A la derecha, Número de nodos expandidos por profundidad en el caso de prueba difícil de Top Spin con 14 Tokens por cada algoritmo de búsqueda.

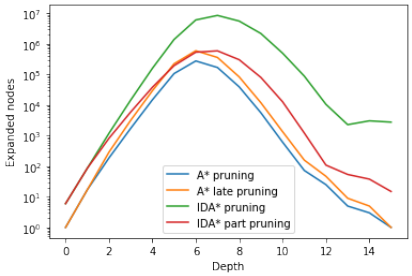
A pesar de que no todo el puzzle cabe en un solo PDB, igualmente los casos de prueba se resuelven en prácticamente nada de tiempo independientemente de su dificultas. Por eso decidimos usar 5 casos de prueba difíciles, cuyos resultados se muestran en la *Tabla 13*. Incluso podemos notar en la *Figura 15* que el número de nodos expandidos por profundidad es del orden de 10^3, mucho menor en comparación a los otros puzzle. Debido a esto, no hay mucho que comparar entre los distintos algoritmos de búsquedas. Todos son tan eficientes que la diferencia entre ellos es prácticamente nula.

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	128.27677	7.48756	3399.53925
A* with Late Pruning	150.89582	7.15296	6054.02018
IDA* with Pruning	383.27356	6.20838	96949.63737
IDA* with Partial Pruning	49.93774	6.20687	49953.60782

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	221.378873	7.51246	2792.71455
A* with Late Pruning	248.904796	7.79342	5289.25927
IDA* with Pruning	289.077547	6.19873	85287.23955
IDA* with Partial Pruning	44.63695	6.20627	39703.65359

Tablas 14 y 15. A la izquierda, promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de prueba fáciles de Top Spin con 17 Tokens por cada algoritmo. A la derecha, tiempo, memoria y número de nodos expandidos en el caso de prueba difícil de Top Spin con 17 Tokens para cada algoritmo de búsqueda.





Figuras 17 y 18. A la izquierda, representación de los PDBs que usamos para Top Spin con 17 Tokens. A la derecha, Número de nodos expandidos por profundidad en el caso de prueba difícil de Top Spin con 17 Tokens por cada algoritmo de búsqueda.

4.8 Top Spin 17 Tokens - Turntable of length 4

Para el Top Spin con 17 Tokens usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 13*. Los 5 casos de pruebas (todos difíciles) escogidos junto a la longitud de sus soluciones fueron

```
10 5 3 16 11 8 1 4 6 13 9 2 15 14 7 12 17 (15)
16 3 14 7 2 5 9 1 12 8 15 10 13 6 11 4 17 (15)
10 16 11 4 12 9 14 1 15 13 3 2 5 8 6 7 17 (15)
9 12 13 1 15 8 3 11 5 14 7 16 4 2 6 10 17 (15)
8 11 4 3 6 7 9 14 13 1 16 5 2 15 12 10 17 (15)
```

Al igual que el puzzle anterior, solo escogimos casos de prueba difíciles pues se resuelven con relativa facilidad, tal como se muestra en la *Figura 14*. Sorprendentemente, el algoritmo con mejor rendimiento y por mucho fue el de IDA* con poda parcial de duplicados, llegando a ser hasta 5 veces más rapido que A* con poda total de duplicados, tal vez sea porque, entre todos los puzzles, este es el que tiene un crecimiento mas elevado y una longitud de soluciones corta. Sin embargo, el que tuvo peor rendimiento fue IDA* con eliminación total de duplicados, incluso podemos notar en la *Figura 18* que el número de nodos que expandió es significativamente mayor que el del resto de los algoritmos.

4.9 Rubik's Cube

Para el Cubo de Rubik usamos como heurística PDBs no aditivos, particionando el puzzle tal como se muestra en *Figura 19*. Los 5 casos de pruebas fáciles escogidos junto a la longitud de sus soluciones fueron

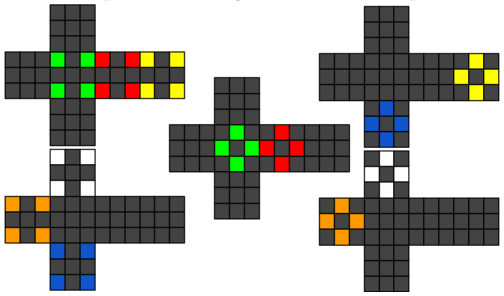
```
O B G G B O G O W W Y B R Y B R B Y W Y O G R B R R G O Y W G R O G Y W W G
R Y B R Y B W W O O (10)

B W G G B B O O Y O W B B Y Y Y W O O R R W B W G R G B Y G W Y Y W R R G G
O Y R G O O W R R B (8)

O O R W B W R O Y Y Y B Y G W G G Y B G R R W B G G R O W W G W B B G B O Y
O R O Y Y O B R W R (8)
```

ALGORITHM	TIME (SECONDS)	MEMORY (GB)	NODES / SECOND
A* with Pruning	0.811613	8.14311	800.8743
A* with Late Pruning	1.041027	8.11364	1005.7376
IDA* with Pruning	0.401234	8.14201	22149.16981
IDA* with Partial Pruning	0.396473	8.02393	11302.15677

Tabla 16. Promedio de tiempo, memoria y número de nodos expandidos en los 5 casos de prueba fáciles del Cubo de Rubik por cada algoritmo de búsqueda.



Figuras 19. Representación de los PDBs que usamos para el Cubo de Rubik

```
R W W R R W R R G G G B G Y G Y B B B O O Y O B B B Y G Y G Y G Y O B O O O
O Y W W W W W R R R (8)

Y R R B W W O G Y G G O O O G B O O O Y B G R W R B G Y B B W G G Y B R B O
W R Y W W R R W Y Y (9)
```

Lamentablemente no pudimos realizar un estudio sobre algún caso de prueba difícil debido a problemas de conectividad. Podemos notar gracias a la *Tabla 16* que para los casos fáciles de este puzzle IDA* es más eficiente que A*, tal como ha sido en la mayoría de los casos de prueba fáciles estudiados en este proyecto. Algo muy importante a destacar es que la generación de nodos por segundo para este puzzle es muy baja en comparación al resto de puzzle. Esto puede deberse al tamaño en la representación de los estados (48 caracteres) y a lo complicada que son las reglas de transición del cubo de rubik, las cuales involucran cambiar el estado de varias posiciones al mismo tiempo, cuando en los otros puzzle en general sólo una pequeña fracción del estado es modificado entre cada movimiento.

5 Detalles de Implementación

5.1 NodesPriorityQueue

La clase `NodesPriorityQueue` tiene 3 campos fundamentales:

- `map<uint64_t, pair<unsigned, Node*>> hash` es un diccionario que mapea las valores de la tabla hash proporcionada por la API de PSVN a pares `{V, N}` donde `V` es el valor del nodo al que apunta `N`.
- `set<pair<unsigned, Node*>> ordered_nodes` es un conjunto de pares `{V, N}` con la misma definición anterior. Cabe destacar que el tipo de dato `set` está implementado en C++ como un arbol rojo-negor, por lo que mantendrá ordenados los nodos según su valor `V`.
- `unsigned (*f) (Node*)` es la función de evaluación de los nodos.

Así, para buscar un nodo según su hash, simplemente tenemos que verificar que se encuentra en el campo `hash`, lo cual es $O(1)$. Mientras que para realizar el reemplazo, primero obtenemos el hash del estado que tiene el nodo, luego, usando `hash` obtenemos el par `{V, N}`, y con ese par, obtenemos el elemento que se encuentra en `ordered_nodes`, y así realizamos el cambio en ambas estructuras en $O(\log n)$.

5.2 InformedSearchs

Para imprimir la memoria virtual usada actualmente se utiliza la estructura `struct sysinfo`, el cual, luego de aplicarle la función `sysinfo`, almacena la memoria RAM y swap usada. Así, solo debemos imprimir la memoria virtual inicial antes de correr el algoritmo y la memoria virtual justo antes de terminar para saber aproximadamente cuanta memoria se usó. Para imprimir el tiempo transcurrido se usó la función `clock()`, marcando el tiempo inicial e imprimiendo su diferencia con el tiempo final.

Las funciones auxiliares `apply_rule` y `revert_rule` pueden parecer redundantes ya que la API de PSVN contiene las funciones `apply_fwd_rule` y `apply_bwd_rule` respectivamente. Sin embargo, estas dos últimas tienen un problema cuando el estado al que se le aplicará la regla y el estado que almacenará el sucesor son el mismo, probablemente porque es modificado mientras es leido por la función. Es por esto que las funciones `apply_rule` y `revert_rule` lo que hacen es generar un estado auxiliar copiando al estado original, y lo usa como estado al que se le aplicará la regla y almacena al sucesor en el estado original. Estas son usadas por IDA* con eliminación parcial de duplicados.

La estructura `NodesPriorityQueue` se usó en A* con eliminación de duplicados, y realiza las funciones de almacenar los nodos ordenados según su valor (costo del camino parcial más la heurística), y permite verificar la existencia de un estado y la sustitución de nodos con el mismo estado de forma eficiente.

Mientras que para A* con eliminación tardía de duplicados se usó una tabla de hash que mapea los valores de hash para los estados dado por la API de PSVN a costos parciales. Así, podemos verificar la existencia de un estado y su costo almacenado de forma eficiente.

Para IDA* con eliminación de duplicados se utilizó una variable de tipo `set` que almacenaba los nodos que se encontraban en el camino actual. Así, solo basta con verificar si un nodo sucesor pertenece a dicho camino para saber si se debe agregar o no.

5.3 PDBs

El proceso de generación de un PDB para un puzzle sigue los siguientes pasos:

1. Compilar el archivo `abstractor.cpp` y `psvn.cpp` de la API de PSVN para obtener un archivo binario `abstractor.out` que nos permita crear la abstracción que necesitamos.
2. Utilizar `abstractor.out` para generar un la abstracción `.psvn` a partir del archivo `.psvn` original y el archivo `abstraction`.
3. Ejecutar `psvn2c` sobre el archivo `.psvn` abstraido para generar un archivo `.c` que contiene las reglas del puzzle abstraido codificadas.
4. Compilar el archivo `dist.cpp` proporcionado por la API de PSVN junto al `.c` del paso anterior para generar un ejecutable `.dist`.
5. Ejecutar el archivo `.dist`, el cual generará un PDB codificado en un archivo tal que cada línea sigue el formato `<VALUE> <STATE>`, donde `<VALUE>` es el costo mínimo desde el estado `<STATE>` hacia el estado objetivo. Este output es almacenado en un archivo `.pdb` que puede llegar a ser muy pesado.
6. Eliminar el archivo `.c` y `.psvn` abstraido y luego copiar y pegar el `.psvn` original en el directorio actual. La razón de hacer esto es que para compilar el siguiente archivo necesitamos usar el psvn con las reglas y estados originales.
7. Compilar el archivo `make_state_map` creado por nosotros el cual inicializa una variable del tipo `state_map_t` proporcionado por la API, y por cada línea del archivo `.pdb` almacena en dicha variable el estado y su valor. Luego de recorrer todo el archivo, almacena la variable de `state_map_t` en un archivo `.state_map`.
8. Ejecutamos `make clean` para quedarnos únicamente con el archivo `.state_map`.

5.4 heuristics

Para evitar crear una función heurística por cada puzzle del mismo tipo pero con diferentes dimensiones, por ejemplo 3 funciones para Top Spin con 12, 14 y 17 tokens respectivamente, donde cada función será casi exactamente igual pero cambiando el valor de algunas variables, decidimos utilizar variables globales que definan el comportamiento de las heurísticas:

- `vector<state_map_t*> pdbs` almacena los PDBs que se usarán en las heurísticas. La función `init_pdbs` permite cargar todos en `pdbs` los archivos `.state_map` que se encuentren en un directorio dado como argumento. Los `.state_map` son cargados en orden alfabético.
- `unsigned (*f) (unsigned, unsigned)` es una función que indica la relación entre heurísticas de bloques PDB, puede tomar el valor de `max_h` para agarrar el máximo en caso de heurísitcas no aditivas, o `sum_h` para sumarlas en caso de heurísitcas aditivas. Las funciones que realizan la asignación de `f` son `set_max` y `set_sum`.
- Cada puzzle tiene su propia variable global `partition` (pudiendo ser de distinto tipo entre cada puzzle) que almacena la forma en que se particionará el puzzle para los PDBs. Es importante que el orden en el que se encuentran los bloques de la partición corresponda al orden en el que son cargados los PDBs en la variable `pdbs`, en caso contrario se obtendrá un bello y hermoso `segmentation fault`. Por cada variante de un puzzle existe una partición correspondiente y una función que realiza la asignación de la variable `partition` global del puzzle genérico a la variable `partition` del puzzle con dimensiones específicas. Por ejemplo, para los NPuzzles existe una variable `string **partition_Npuzzle` y para el 15Puzzle y 24Puzzle están las variables `string partition_15puzzle[4][4]` y `string partition_24puzzle[5][5]` junto a las funciones `set_15puzzle` y `set_24puzzle` respectivamente que asignan a `partition_Npuzzle` la partición que corresponda.

Además, cada puzzle tiene su propia función `make_state_abs` que toma un estado y un bloque de la partición y genera un nuevo estado abstraido según dicho bloque. Una vez con estos elementos, las heurísticas de cada puzzle siguen el mismo comportamiento: Reciben un esto, recorren el vector `pdbs` y por cada uno generamos un estado abstraido dado el bloque de la partición correspondiente, obtenemos el valor de dicho estado según el `state_map` y actualizamos el valor de la heurísitca según `f`.

6 Conclusión

El rendimiento de A* e IDA* depende mucho del tipo de puzzle que querramos resolver y de la dificultad del caso de prueba. Tal como vimos con las Torres de Hanoi, complicar un poco un estado inicial puede llevar a que un algoritmo pase de correr en segundos a tardar cerca de una hora. Sin embargo, podemos hacer algunas generalizaciones para saber cuando hay que usar A* o IDA*. La primera es verificar la longitud de las soluciones promedios de un puzzle, mientras mas largo sea, peor será el rendimiento de IDA*, pues este deberá recorrer todo el espacio de búsqueda hasta cada profundidad. Siguiendo la misma lógica, al ser más difícil un caso de prueba, generalmente significa que se necesitan más pasos para resolverse, lo que termina perjudicando a IDA*. Ahora, si las longitudes de las soluciones es corta, IDA* es la mejor opción, tal como vimos en el análisis de Top Spin.

Respecto a los PDBS, vimos lo útiles que pueden llegar a ser, sobre todo cuando es posible almacenar gran parte del puzzle en un solo PDB. Además, es importante saber como separar en bloques al puzzle, pues una buena elección resultará en una aceleración en la resolución de los problemas, así como en las Torres de Hanoi la heurística logró orientar a IDA* directamente a la solución una vez los bloques más pesados estuvieron en la posición correcta. Esto no hubiera sido posible si, por ejemplo, en lugar de escoger un bloque de PDB que no incluyera los discos mas pesados, no se incluyeran algunos discos livianos, entonces la heurística no podría guiar al algoritmo directamente a la solución hasta que dichos discos estuvieran ordenados, pero al alcanzar ese punto o el puzzle ya esta casi resuelto, o se tendrán que desordenar para darle paso a los discos pesados.

Hace falta hacer más pruebas pues, debido a problemas de conectividad, no logramos hacer todos los casos de prueba que queriamos analizar. Principalmente para los casos de prueba difíciles, los cuales solo pudimos analizar uno por puzzle (o ninguno en caso del Cubo de Rubik), por lo que pudimos haber estado estudiando un caso de prueba anormal, el cual daría datos anormales no aptos para la generalización.