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 XX-XXXXX, Wilfredo Graterol XX-XXXXX

Búsqueda Informada: A* vs IDA*

June 12, 2021

1 Introducción

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/...
|__ 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

- `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`.
- `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 Casos de Prueba

3.1 15 Puzzle

Para el *15 Puzzle* usamos como heurística la distancia Manhattan.

```
STATE (EASY):   14 1 9 6 4 8 12 5 7 2 3 B 10 11 13 15
|           FUNCTION|   TIME (SEC)|  MEMORY (GB)|       NODES/SEC|    SOL-LEN|
|                 A*|   172.613782|      5.39361|    138210.58622|         45|
|         A* pruning|     2.627228|      0.04398|     62283.51708|         45|
|    A* late pruning|     2.593914|      0.04016|     90053.48674|         45|
|               IDA*|   294.174137|      0.00055|    243362.32182|         45|
|       IDA* pruning|     1.537629|       0.0129|    410535.96154|         45|
|  IDA* part pruning|     2.629333|      0.00024|    240086.36411|         45|
```

```
STATE (MEDIUM): 12 9 B 6 8 3 5 14 2 4 11 7 10 1 15 13
|           FUNCTION|   TIME (SEC)|  MEMORY (GB)|       NODES/SEC|    SOL-LEN|
|                A*|                    TOO MUCH MEMORY                       |
|         A* pruning|   361.748725|      2.56905|     26938.20137|         50|
|    A* late pruning|   184.561287|      2.34054|     85301.65917|         50|
|               IDA*|                     TOO MUCH TIME                       |
|        IDA* pruning|   243.698492|      0.00148|    413353.29231|         50|
|  IDA* part pruning|   421.172057|       0.0001|    239603.34577|         50|

STATE (HARD):   5 9 13 14 6 3 7 12 10 8 4 B 15 2 11 1
|           FUNCTION|   TIME (SEC)|  MEMORY (GB)|       NODES/SEC|    SOL-LEN|
|                A*|                    TOO MUCH MEMORY                       |
|         A* pruning|   633.547779|       4.1973|     25436.99234|         57|
|    A* late pruning|   305.977159|      3.92621|     77683.54042|         57|
|               IDA*|                     TOO MUCH TIME                       |
|        IDA* pruning|   349.547181|      0.00082|    404336.85832|         57|
|  IDA* part pruning|   591.625852|      0.00195|    238931.14461|         57|
```

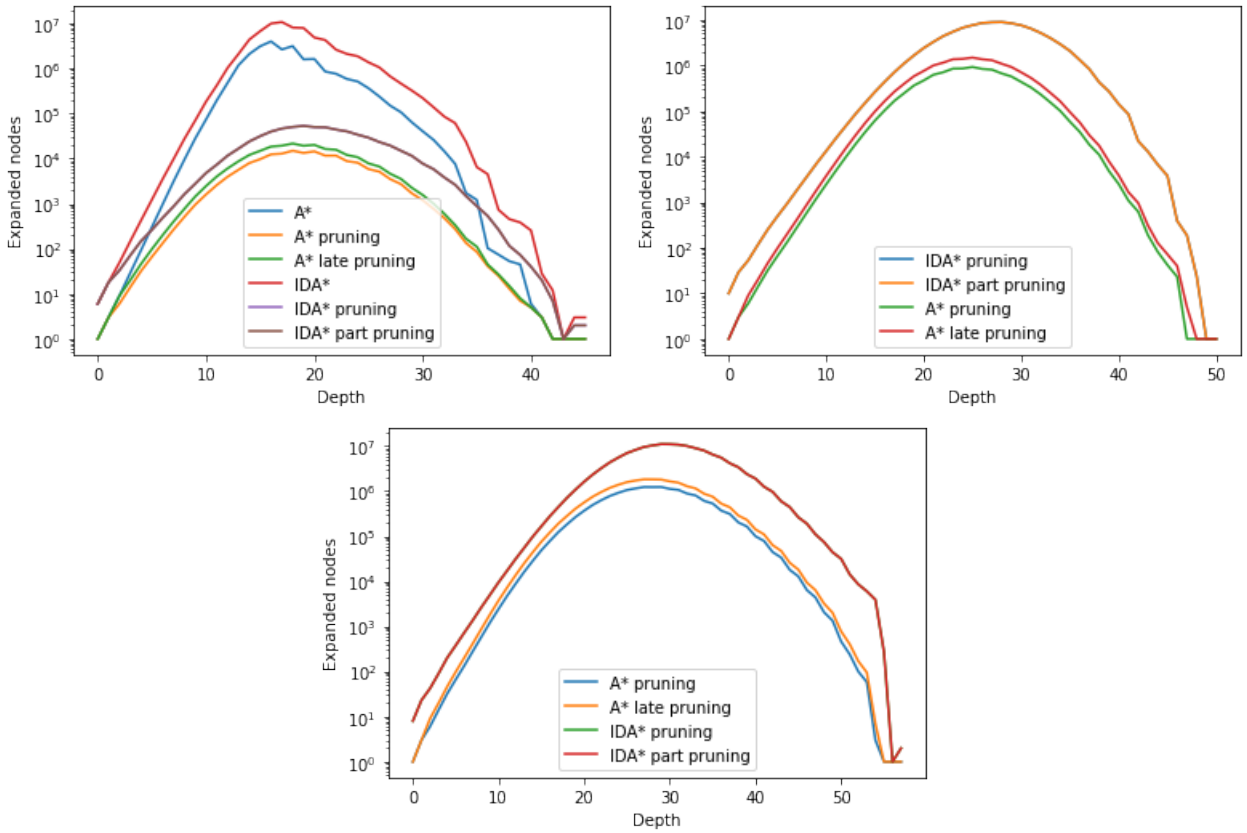

Figure 1: Left, easy. Right, medium. Down, hard

3.2 24 Puzzle

3.3 Towers of Hanoi 4 Pegs - 12 Disks

3.4 Towers of Hanoi 4 Pegs - 14 Disks

3.5 Towers of Hanoi 4 Pegs - 18 Disks

3.6 Top Spin 12 Tokens - Turntable of length 4

3.7 Top Spin 14 Tokens - Turntable of length 4

3.8 Top Spin 17 Tokens - Turntable of length 4

3.9 Rubik's Cube

4 Detalles de Implementación

4.1 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 $O(\log n)$.

4.2 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.

4.3 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`.

4.4 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`.

5 Conclusión