Informe de Taller 1:
Generalización del problema de los bidones de agua

Nombre: Ignacio Solar Arce
Profesor: Pablo Román Asenjo
Sección: B-2
Fecha: 4/11/2024

1. Explicación breve del Programa

En este informe se presenta la estructuración de un programa que permite resolver el problema de los bidones de forma generalizada.

La generalizacion del problema de los bidones de agua se ve en base a la cantidad total de bidones que se pueden utilizar para llegar a un cierto objetivo. El programa utiliza el algoritmo A*, que en vez de ir buscando todos las combinaciones de bidones, sigue su camino en base a heuristicas.

Las heuristicas son mecanismos que permiten "ponderar" o "pesar" distintas combinaciones de bidones y decidir si "vale la pena" o no explorar por ese trayecto, esta información resulta fundamental, ya que permite disminuir la búsqueda de manera exponencial. Las heurísticas implementadas, estas tienen que ser admisibles, osea, para este problema, que generen un valor heuristico menor mientras mas me acerco a la solución, siendo el menor cuando estoy en la solucion.

2. Sistema de Heurísticas y Optimización

2.1. Heurísticas Principales

El sistema utiliza dos heurísticas principales que se combinan para evaluar cada estado:

2.1.1. Pattern Value

Esta heurística evalúa las coincidencias exactas con el estado objetivo:

- Se calcula un valor base para cada bidón que coincide con el objetivo
- El valor se pondera según la posición del bidón (preferencia por coincidencias a la izquierda)
- Se calcula como:

$$pattern\_boost = w_{explore} \cdot 30{,}0 + w_{balance} \cdot 25{,}0 + w_{optimize} \cdot 20{,}0 \tag{1}$$

- Para cada coincidencia se aplica un bonus por posición:

$$position\_factor = 1{,}0 - \frac{posicion}{total\_bidones} \tag{2}$$

- El valor final para cada coincidencia es:

$$pattern\_value += pattern\_boost \cdot (1{,}0 + position\_factor \cdot 0{,}5) \tag{3}$$

2.1.2. Transfer Value

Esta heurística estima el costo mínimo de operaciones necesarias:

- Para cada bidón que no coincide, calcula la diferencia con el objetivo
- Considera el máximo valor disponible en el segmento actual

- Estima operaciones mínimas necesarias:

$$operations = \frac{|diferencia|}{max\_local} \tag{4}$$

- Pondera según las estrategias y momentum:

$$\begin{aligned} transfer\_factor = {} & w_{explore} \cdot (10{,}0 - 5{,}0 \cdot momentum_{combined}) + \\ & w_{balance} \cdot (20{,}0 - 10{,}0 \cdot momentum_{combined}) + \\ & w_{optimize} \cdot (30{,}0 - 15{,}0 \cdot momentum_{combined}) \end{aligned} \tag{5}$$

- El valor final por bidón es:

$$transfer\_value+ = operations \cdot transfer\_factor \tag{6}$$

2.2. Penalización por Profundidad

La penalización por profundidad es el mecanizmo crucial que permite evitar soluciones largas:

- Se calcula considerando tres factores:
  - Profundidad actual del estado
  - Tamaño del problema (número de bidones)
  - Momentum global (progreso hacia la solución)
- La fórmula base es:

$$depth\_penalty = min(0{,}1 + \frac{depth}{size \cdot 3{,}0} \cdot (0{,}8 + momentum_{global}), 0{,}3) \tag{7}$$

- Mantiene como mínimo un 10 % de penalización hasta un máximo del 30 %, escalando según el tamaño del problema. Tiene importancia considerar esto debido a que permite reducir la cantidad de patterns encontrados(ponderado) y al aumentar el costo mientras mas profundo sea, favorece a las soluciones cortas que mantiene un buen progreso sin sacrificar optimalidad.

2.3. Análisis por Segmentos

El sistema divide los estados en segmentos de 8 bidones para un análisis más general. Para cada segmento se mantiene:

- Un contador de coincidencias con el estado objetivo
- El valor máximo presente en el segmento
- Un factor de momentum local

La segmentación permite tener un análisis sobre los subproblemas que se encuentren en el estado, estimar de mejor forma la cantidad de operaciones que sean necesarias y también detectar con anterioridad si se crean patterns correctos, que no necesariamente es útil en las etapas iniciales de la solucion.

2.4. Sistema de Momentum

2.4.1. Momentum Global

El momentum habla sobre cuan cerca se esta de la solución, un valor entre 0 y 1 aproxima esto. Tiene la utilidad de permitir ponderar factores en base a la cercanía a la solución, agregando otro factor ponderador junto a la profundidad. Se calcula como:

$$momentum_{global} = 1{,}0 - \frac{bidones\_coincidentes}{total\_bidones} \tag{8}$$

2.4.2. Momentum por Segmento

El sistema se divide en segmentos de 8 bidones, se hace una pasada inicial donde se divide y Para cada segmento:

$$momentum_{segmento} = 1{,}0 - \frac{coincidencias\_segmento}{size\_segmento} \tag{9}$$

2.4.3. Momentum Combinado

$$momentum_{combinado} = 0{,}7 \cdot momentum_{global} + 0{,}3 \cdot momentum_{segmento} \tag{10}$$

2.5. Estrategias de Búsqueda

El sistema implementa tres estrategias que balancean la estrategia que se utilizara. Los pesos de cada una de estas se calculan de manera dinámica en función de la profundidad, tamaño del problema y también en base a el desempeño reciente, en mejorar consecutivas o en estancamientos.

Este sistema es extremadamente útil, en base a la detección de estancamientos, podemos priorizar una exploración de nuevos estados en vez de optimizar los estados cercanos, o también al revés, si estamos explorando demasiado y no se acerca mucho a la solución, se podría potenciar la optimización para intentar cambiar eso. Ya que se mantienen muchos sistemas distintos, es necesario normalizar todo entre 1 y 0, los pesos iniciales pueden variar respecto a la profundidad o momentum.

2.5.1. Estrategia de Exploración

Busca expandir el espacio de búsqueda y encontrar nuevos estados Características:

- Peso inicial: 0.6
- Decae linealmente con la profundidad
- Mínimo: 0.25
- Factor de exploración: $(30{,}0 - 15{,}0 \cdot momentum_{combinado})$

2.5.2. Estrategia de Balance

Mantiene un equilibrio entre explorar por nuevas soluciones u optimizar las existentes. Características:

- Peso constante: 0.3
- Factor de balance: $(25{,}0 - 12{,}5 \cdot momentum_{combinado})$
- No varía con la profundidad, siempre se requiere mantener un balance en el sistema.
- Actúa como mediador entre exploración y optimización

2.5.3. Estrategia de Optimización

En vez de intentar encontrar nuevas soluciones, esta estrategia se preocupa de refinar y mejorar las soluciones que ya existen. Características:

- Peso inicial: 0.2
- Crece linealmente con la profundidad
- Máximo: 0.6
- Factor de optimización: $(20{,}0 - 10{,}0 \cdot momentum_{combinado})$

2.6. Sistema de Adaptación

La estructura `strategy_weights` es la que almacena los parámetros para poder ir cambiando dinamicamente la prioridad de cada estrategia.

2.6.1. Factores de Ajuste

El sistema ajusta los pesos según:

- Mejoras consecutivas
- Períodos de estancamiento
- Tamaño del problema
- Profundidad actual

2.6.2. Reglas de Adaptación

La estructura `adaptive_params` mantiene las variables para ajustar las ponderaciones en base al desempeño, esto se expresa como mejoras consecutivas o estancamientos (plateaus)

- Tras 3 mejoras consecutivas: aumenta optimización
- Tras 2 períodos de estancamiento: aumenta exploración
- Factor de tamaño: $min(1{,}0, \frac{size\_problema}{30})$
- Factor de profundidad: $min(1{,}0, \frac{profundidad}{60})$

2.6.3. Reglas de Adaptación

La estructura `adaptive_params` mantiene las variables para ajustar las ponderaciones en base al desempeño, esto se expresa como mejoras consecutivas o estancamientos (plateaus)

- Tras 3 mejoras consecutivas: aumenta optimización
- Tras 2 períodos de estancamiento: aumenta exploración
- Factor de tamaño: $min(1{,}0, \frac{size\_problema}{30})$
- Factor de profundidad: $min(1{,}0, \frac{profundidad}{60})$

2.6.4. Sistema de Simulated Annealing (Temperatura)

El sistema de Simulated Annealing se incorpora como una metaheurística[1] que complementa el sistema de calculo heuristico. Este método se basa en la analogía con el proceso físico de recocimiento de materiales (annealing), donde se lleva un sólido a altas temperaturas y luego se enfría gradualmente, en donde se observa la rigidez o flexibilidad de la estructura atómica del material [1].

Para el problema, la "temperatura.[es] el parámetro que decide la probabilidad de aceptar soluciones que temporalmente degradan el acercamiento al objetivo. A temperaturas altas, se permite explorar libremente el espacio de soluciones, incluso aceptando movimientos que empeoren la situación actual. A medida que la temperatura disminuye, el algoritmo se vuelve más rígido(selectivo), favoreciendo gradualmente solo las soluciones que mejoran el objetivo.

Para aceptar estados peores, se rige mediante el criterio de Metrópolis[2]: una nueva solución que degrada el objetivo se acepta con una probabilidad $e^{\frac{f(i)-f(j)}{c}}$, donde $f(i)$ y $f(j)$ son los valores de la función objetivo para el estado actual y nueva respectivamente, y c es la temperatura actual. La gran finalidad de esto es la posibilidad de escapar de mínimos, especialmente en la etapa exploratoria de la búsqueda, mientras que en las etapas finales, con temperaturas más bajas, el algoritmo converge hacia un óptimo, refinando la mejor solución encontrada.

- **Temperatura Inicial**:

$$T_0 = 1{,}0 \text{ (INITIAL\_TEMPERATURE)} \tag{11}$$

- **Actualización de Temperatura**:
  - En caso de mejora:

$$T_{new} = T_{current} \cdot 0{,}95 \tag{12}$$

  - En caso de estancamiento:

$$T_{new} = min(T_{current} \cdot 1{,}3, T_0) \tag{13}$$

  - Temperatura mínima:

$$T_{min} = 0{,}05 \tag{14}$$

- **Criterio de Aceptación**: Para estados con peor heurística:

$$P(aceptar) = e^{-\frac{\Delta E}{T \cdot 0{,}1}} \tag{15}$$

  donde ΔE es la diferencia relativa en el valor heurístico:

$$\Delta E = \frac{weight_{new} - weight_{current}}{weight_{current}} \tag{16}$$

- **Generación de Estados Aleatorios**:
  - Número de estados a generar:

$$states = max(3, min(random\_states\_per\_check \cdot \frac{T}{T_0}, random\_states\_per\_check)) \tag{17}$$

  - Solo se consideran estados con $\Delta E < 0{,}2$ para el criterio de aceptación

Este sistema de temperatura trabaja en conjunto con ambas estructuras adaptativas del sistema heuristico y de busqueda poder lograr lo siguiente:

- Permitir exploración temprana cuando la temperatura es alta, agregando mas estados.
- Favorecer optimización local cuando la temperatura baja, evitando agregar mas estados.
- Reactivar exploración si se detecta estancamiento prolongado, en conjunto a los parámetros de stagnation.
- Ajustar dinámica mente la agresividad de la búsqueda según el progreso

2.7. Cálculo del Peso Final

Finalmente, luego de combinar cada una de las estrategias, estas se ponderan junto a las penalizaciones para generar un peso final. Si bien no se presenta, el sistema de estrategias es ponderado en base a los factores de cada una de las heuristicas utilizadas.

$$\begin{aligned} peso = {} & valor\_transferencia \cdot (1{,}5 + momentum_{global}) + \\ & valor\_patron \cdot (1{,}0 - penalizacion_{profundidad}) + \\ & profundidad \cdot penalizacion_{profundidad} \cdot (10{,}0 + 20{,}0 \cdot momentum_{global}) \end{aligned} \tag{18}$$

Donde:

- $valor\_transferencia$: Costo estimado de operaciones necesarias
- $valor\_patron$: Bonificación por coincidencias exactas
- $penalizacion_{profundidad}$: Factor que crece con la profundidad
- $momentum_{global}$: Indicador de progreso general

2.8. Robin Hood Hashing

El ordenamiento de la tabla hash se basa en la idea de Robin hood, donde al intentar insertar se considera un psl (Probe sequence length), que es cuantas colisiones genera antes de insertar correctamente. La tabla hash se aprovecha de esta información "robando" la posición de algún valor que tenga un menor psl y luego insertar ese valor a la tabla hash[3], "dándole a los pobres las cosas de los ricos".

Mediante datos medidos por el profiler Tracy, se encontró que casi nunca es necesario tener que robar posiciones debido a la distribución que genera la función hash, generando psl no mas de 7 para una tabla hash de 3.4 millones de elementos, esto se aprecia en el anexo4. Ademas, la tabla hash considera un factor de balance, respecto al porcentaje de llenado o si bien se genero un psl horrendo para cierta situación, ambas ameritan incrementar el tamaño de la tabla, que a su vez es una potencia de 2, esto permite utilizar un bithack para evitar utilizar el modulo y usar un AND lógico entre secuencias de bits para encontrar el modulo.

2.9. MurMur3 32u hash

MurMur es un suite de funciones hash que tienen la peculiaridad de ya estar benchmarkeadas en otras situaciones, para el caso de uso, 32 bits de largo resulta suficiente. La función tiene excelente distribución de números y ademas muy veloz, integrando bit mixes y rotaciones, terminando con shifts y valores primos. Para el caso de uso, en un ejemplo de 15 bidones, se encontró mediante Tracy que en promedio su calculo es de 18 nanosegundos, siendo ejecutado 3.4 millones de veces.

2.10. Pairing Heap

Para este problema, el Heap se considera el conjunto abierto, para ciertas situaciones esto puede llegar a tener millones de estados, en el caso de 15 bidones, se encontraron almenos 3.2 millones de estados en el conjunto abierto.

Pairing Heap a comparación de un heap binario tiene una ventaja en agregar elemento a si mismo, ademas que la combinación de push y pop también se vuelve mas eficiente debido a que Pairing solo almacena punteros para cada uno.

Sucede mucho la necesidad de hacer merge del heap, en la cual se decide cual mantener como padre, en vez de utilizar downkey, merge solo hace cambios de punteros recursivamente[4].

La función merge es la mas llamada en la ejecución de un programa, en el caso de 15 bidones se ejecuta aproximada 32 millones de veces, con un promedio por llamada de 56 nanosegundos. Si bien es uno de los mayores limitantes en el tiempo, es increíblemente eficiente, su único limitante, que tampoco puede controlar, son los estados que se le agregan a si mismo.

A comparación de la implementación original, ajustar el heap llega a ser hasta 10 veces mas rápido que antes por ejecución, aunque si bien ahora se ejecute muchas mas veces, la relación tiempo promedio por ejecución es mucho menor.

2.11. Variación de parámetros vía Tracy Profiler

Se mantienen muchos parámetros distintos dentro de la ejecución, que permiten modificar la actitud de la heuristica. Al tener el perfilado y los valores de cada solución que se encuentra, se modifican los parámetros para llegar a un comportamiento que en general prefiere explorar sobre quedarse en soluciones especificas. Esto tiene la consecuencia de generar una cantidad mayor de estados en general y por consecuente también el tiempo de ejecución, pero logra encontrar soluciones mas precisas. Como ejemplo, la modificación de parámetros en un problema de 20 bidones con valores primos permitió bajar de 510 pasos en 300 ms a 57 pasos en 4000ms.

3. Aspectos de implementación y eficiencia

3.1. Robin Hood Hashing

El ordenamiento de la tabla hash se basa en la idea de Robin hood, donde al intentar insertar se considera un psl (Probe sequence length), que es cuantas colisiones genera antes de insertar correctamente. La tabla hash

se aprovecha de esta información "robando" la posición de algún valor que tenga un menor psl y luego insertar ese valor a la tabla hash, "dandole a los pobres las cosas de los ricos".

Mediante datos medidos por el profiler Tracy[5], se encontró que casi nunca es necesario tener que robar posiciones debido a la distribución que genera la función hash, generando psl no mas de 7(extremadamente bueno) para una tabla hash de 3.4 millones de elementos. Ademas, la tabla hash considera un factor de balance, respecto al porcentaje de llenado o si bien se genero un psl horrendo para cierta situación, ambas ameritan incrementar el tamaño de la tabla, que a su vez es una potencia de 2, esto permite utilizar un bithack para evitar utilizar el modulo y usar un AND lógico entre secuencias de bits para encontrar el modulo[6].

3.2. MurMur3 32u hash

MurMur es un suite de funciones hash que tienen la peculiaridad de ya estar benchmarkeadas en otras situaciones[7], para el caso de uso, 32 bits de largo resulta suficiente. La función tiene excelente distribución de números y ademas muy veloz, integrando bit mixes y rotaciones, terminando con shifts y valores primos. Para el caso de uso, en un ejemplo de 15 bidones, se encontró mediante Tracy que en promedio su calculo es de 18 nanosegundos, siendo ejecutado 3.4 millones de veces, esto se puede apreciar en el Anexo 3.

3.3. Pairing Heap

Para este problema, el Heap se considera el conjunto abierto, para ciertas situaciones esto puede llegar a tener millones de estados, en el caso de 15 bidones, se encontraron al menos 3.2 millones de estados en el conjunto abierto, esto se puede apreciar en el Anexo 1.

Pairing Heap a comparación de un heap binario tiene una ventaja en agregar elemento a si mismo, ademas que la combinación de push y pop también se vuelve mas eficiente debido a que Pairing solo almacena punteros para cada uno.

Sucede mucho la necesidad de hacer merge del heap, en la cual se decide cual mantener como padre, en vez de utilizar downkey, merge solo hace cambios de punteros recursivamente.

La función merge es la mas llamada en la ejecución de un programa, en el caso de 15 bidones se ejecuta aproximada 32 millones de veces, con un promedio por llamada de 56 nanosegundos, esto se puede apreciar en el Anexo 2. Si bien es uno de los mayores limitantes en el tiempo, es increíblemente eficiente, su único limitante, que tampoco puede controlar, son los estados que se le agregan a si mismo.

A comparación de la implementación original, ajustar el heap llega a ser hasta 10 veces mas rápido que antes por ejecución, aunque si bien ahora se ejecute muchas mas veces, la relación tiempo promedio por ejecución es mucho menor.

3.4. Variación de parámetros vía Tracy Profiler [5]

Se mantiene una cantidad grande de parámetros para la heuristica, gracias al profiler Tracy se puede saber en verdad como evolucionan en el tiempo cada una de ellas, editando la velocidad de convergencia, la calidad de las soluciones, uso de recursos debido a la necesidad de explorar u optimizar los estados que se mantienen. Esto tiene la consecuencia de que es posible varias los parámetros para poder generar que se acerque al optimo global lo mas que pueda, para este caso de uso, se utilizo para modificar la búsqueda a una que priorice la cantidad menor de estados para llegar a la solución. Esto tiene la consecuencia de generar una cantidad mayor de estados generados totales, como ejemplo, variar la penalización en la profundidad, para el mismo problema el sistema puede generar 2 millones de estados con una cantidad de pasos decente o hasta alrededor de 10 millones con una cantidad de pasos casi optima.

4. Ejecución del código

Para la ejecución del código se debe utilizar el makefile asociado para compilar el programa, existe las opciones: clean, all y tracy. Para activar la compatibilidad con el perfilador Tracy se debe compilar con Tracy en vez de all, en el caso que se compile con all, las variables definidas para el profiler estan protegidas por un `idef` block en `TracyMacros.h`.

Una ejecución completa se vería de esta forma:

- ejecutar `make clean all`
- ejecutar ./water_jugs
- Se llegara al menú de selección, ingresar archivo, solucionar, ejecutar los tests o salir.
- En el caso de ingresar un archivo, los de ejemplo se encuentras dentro de la carpeta examples, para usar se debe agregar el prefijo examples/file.txt
- Luego se decide solucionar, si es que se puede solucionar el programa retornada la solución, el tiempo y los pasos para llegar a el.

5. Anexos

Figura 1: Size del heap para un problema de 15 bidones

Name	Total time	Counts	MTPC
main	3.73 s (30.23%)	1	3.73 s
merge	1.83 s (14.82%)	32,194,739	56 ns
PairingHeap_mergePai	1.82 s (14.76%)	17,214,904	105 ns

Figura 2: Estadisticas para el heap en un problema de 15 bidones

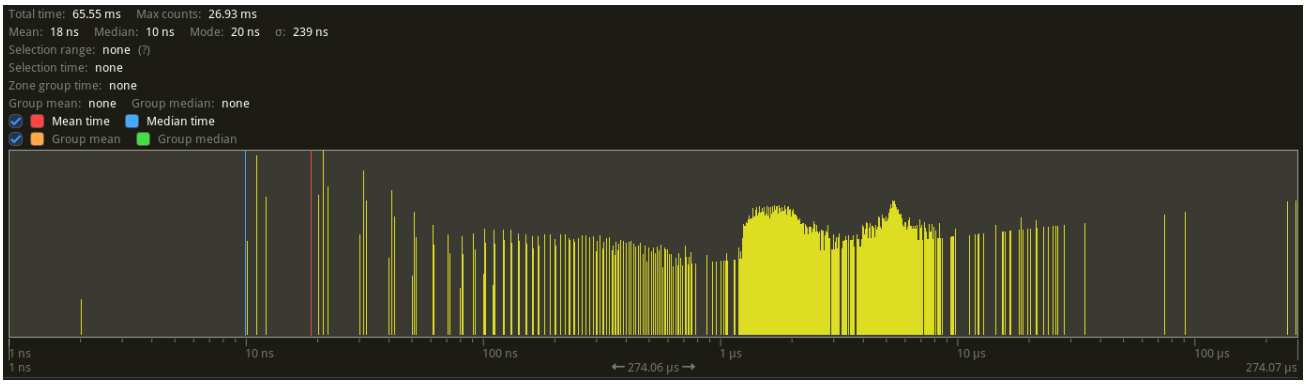

Figura 3: Estadisticas de performance de murmurhash

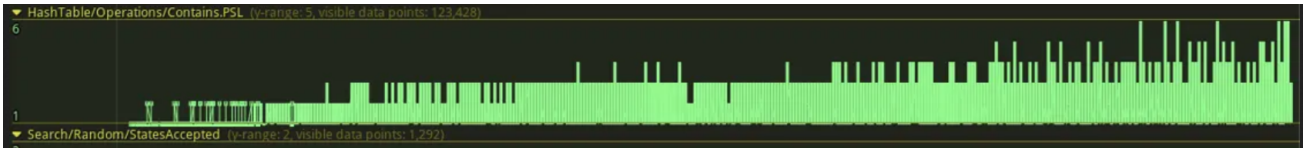

Figura 4: Estadisticas de la cantidad de colisiones para un problema

Referencias

[1] C. de Vecchi, S. Espié, F. Vienne, and H. Mohellebi, "Influence of static obstructions on sight distance and overtaking on two-lane roads," https://enac.hal.science/hal-01887543/document, 2018, hAL Id: hal-01887543. [Online]. Available: https://enac.hal.science/hal-01887543/document

[2] C. R. Aragon, "Simulated annealing: Part 1," 2005, accessed: 2024-11-07. [Online]. Available: https://faculty.washington.edu/aragon/pubs/annealing-pt1a.pdf

[3] C. University, "Robin hood hashing," 2023, accessed: 2024-11-07. [Online]. Available: https://www.cs.cornell.edu/courses/JavaAndDS/files/hashing_RobinHood.pdf

[4] R. Sedgewick, "Heaps," 2011, princeton University, Accessed: 2024-11-07. [Online]. Available: https://www.cs.princeton.edu/courses/archive/spr11/cos423/Lectures/Heaps.pdf

[5] Wolfpld, "Tracy: A profiler for c++," 2024, gitHub repository, Accessed: 2024-11-07. [Online]. Available: https://github.com/wolfpld/tracy

[6] S. Anderson, "Bit hacks," 2015, stanford University, Accessed: 2024-11-07. [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html

[7] A. Appleby, "Smhasher and murmurhash," 2008, gitHub repository, Accessed: 2024-11-07. [Online]. Available: https://github.com/aappleby/smhasher/tree/master