TP1 – Asignación de recursos en la nube

Josefina Jahde, Dafydd Jenkins, Candelaria Sutton

Septiembre, 2024

Introducción

En la actualidad, la asignación eficiente de recursos en la nube, particularmente de GPUs, es un desafío crucial para proveedores de servicios y usuarios que dependen de recursos de alto rendimiento. Las GPUs son fundamentales para ejecutar aplicaciones intensivas en cómputo, como el entrenamiento de modelos de inteligencia artificial, la renderización de gráficos, o la simulación científica. Sin embargo, la demanda de estas instancias varía en términos de cantidad, capacidad y duración.

El problema de asignación de recursos en la nube consiste en distribuir de manera óptima un conjunto limitado de GPUs entre las múltiples solicitudes de los clientes. Cada solicitud tiene diferentes requerimientos en cuanto a capacidad de cómputo, duración de uso y prioridad. Un mal manejo de estos recursos puede derivar en tiempos de espera elevados, costos innecesarios o la subutilización de las GPUs.

Resolver este problema es clave para garantizar que los recursos se usen de manera eficiente, maximizando el beneficio tanto para los proveedores de la nube como para los clientes, que buscan un rendimiento óptimo a costos razonables.

En este trabajo se busca resolver dicho problema de asignación de GPUs a clientes de servidores en la nube. El objetivo del trabajo será desarrollar distintos algoritmos que resuelvan el problema descrito. A su vez, se realizarán distintos experimentos para medir la eficiencia de cada algoritmo en relación a la cantidad de instancias cliente, de máquinas físicas disponibles, la capacidad de las mismas y el lenguaje de programación en el cual se implementó el algoritmo

En este escrito se pueden hallar las siguientes secciones con temáticas a tratar:

- Descripción del problema: descripción formal del problema planteado
- Modelado del problema: cómo se modeló la resolución del problema a partir de otros problemas conocidos.
- Algoritmos desarrollados: descripción de cada uno de los algoritmos desarrollados.
- Experimentación, resultados y discusión: descripción de los distintos experimentos llevados a cabo, exposición de los resultados obtenidos y discusión de las implicancias de los mismos.

Descripción del problema

Formalmente, el problema se puede definir como la siguiente maximización:

$$\text{máx} \sum_{i=1}^{n} \sum_{j=1}^{m} b_i \cdot x_{ij} \tag{1}$$

$$\text{sujeto a} \sum_{i=1}^{n} g_i \cdot x_{ij} \leq G_j \qquad \forall j = 1, \ldots, m \tag{2}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad \forall i = 1, \ldots, n \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i = 1, \ldots, n, j = 1, \ldots, m \tag{4}$$

(1) indica la función a maximizar, que es la suma de los beneficios por cada instancia asignada a una máquina. En (2) se garantiza que la cantidad de instancias asignadas a la máquina j sea menor o igual a la cantidad de GPUs disponibles en la misma. Con la desigualdad (3) se establece que cada instancia será asignada como mucho a una máquina. (4) establece que las variables de decisión son binarias.

A continuación se presenta un ejemplo para ilustrar mejor el problema:

Se cuenta con dos máquinas de 5 GPUs cada una. A su vez, se cuenta con instancias con pesos = [3,4,5,1] y beneficios = [5,3,3,1]. En este caso, el máximo beneficio obtenible es 9 y una de las posibles soluciones que generan dicho beneficio es la primera instancia en una máquina y la segunda y la cuarta en otra. La tercera no se asigna a ninguna máquina.

Modelado

Para la resolución de este problema, nos basamos en algoritmos conocidos para la resolución del problema knapsack. Decidimos pensar en cada máquina física como una mochila, y a cada instancia cliente como un ítem a meter en la mochila. Pensando de esa manera, lo que se tienen son un conjunto de elementos y muchas mochilas donde guardarlos. Por eso, adaptamos los algoritmos desarrollados serán variaciones de los que resuelven el problema knapsack.

Los algoritmos de fuerza bruta y backtracking utilizarán "soluciones" S. Las mismas serán vectores de enteros que indicarán en su posición i la máquina a la que se asignó el instancia i. Una instancia asignada a la máquina 0 será una instancia que no se asigna a ninguna máquina. De esta forma, los elementos de S pertenecerán a {0, 1, …, n}, donde n es la cantidad de máquinas.

Igualmente, el algoritmo de reconstrucción de la solución luego de utilizar el algoritmo de programación dinámica devolverá una solución S como las descritas anteriormente.

Algoritmos desarrollados

Se desarrollaron cuatro algoritmos para la resolución del problema; uno de fuerza bruta, dos basados en técnicas de backtracking (poda por optimalidad y factibilidad) y uno de programación dinámica. Los cuatro algoritmos luego fueron implementados en C++. Adicionalmente, el algoritmo de programación dinámica fue implementado en Python.

A continuación se describirá cada uno de los algoritmos desarrollados junto con consideraciones a tener en cuenta para cada uno.

Fuerza bruta

Este algoritmo utiliza recursión y parte de una solución S vacía.

Como caso base, cuando la longitud de S alcanza la cantidad de instancias, si la solución es factible, es decir no hay máquinas que tengan más GPUs asignadas de las que tiene disponibles, y el beneficio conseguido con esa solución es mejor que el conseguido con la mejor solución almacenada, se almacena la nueva mejor solución y se retorna el beneficio de la solución. Si no, se retorna 0.

Si la longitud de S es menor a la cantidad de instancias (es decir, queden instancias por asignar), para cada m perteneciente a {0, 1, …, n} se busca el máximo beneficio obtenible al agregar a la solución S actual el elemento m (es decir, al asignar la siguiente instancia a la máquina m). Para esto, se llama recursivamente a la función con S = S U {m}. De esta forma, en algún momento se llega al caso base de la recursión y se obtiene un beneficio. Cuando se obtiene el beneficio máximo para cada posible m, se retorna dicho beneficio. Si no, se retorna 0.

Pseudo-código:

```
llenar_maquinas_FB (beneficios, S, best_solution):
    if (|S|  == |beneficios|):
        if(S es factible y beneficio de S > beneficio de best_solution):
            best_solution = s
            return suma_beneficios(s)
        end if
        else:
            return 0
        end else
    end if
```

```
    else:
        valores = []
        for (m perteneciente a {0, 1, …, n}):
            S = S U {m}
            beneficio = llenar_maquinas_FB(beneficios, S, best_solution)
            agregar beneficio a valores
        end for
        return maximo(valores)
end else
```

Backtracking - Optimalidad

Este algoritmo es recursivo y se basa en el de fuerza bruta.

Como caso base, cuando la longitud de S alcanza la cantidad de instancias, si la solución es factible, es decir no hay máquinas que tengan más GPUs asignadas de las que tiene disponibles, y el beneficio conseguido con esa solución es mejor que el conseguido con la mejor solución almacenada (best_solution), se almacena la nueva mejor solución y se retorna el beneficio de la solución. Si no, se retorna 0.

Si la longitud de S es menor a la cantidad de instancias (es decir, queden instancias por asignar), para cada m perteneciente a {0, 1, …, n} se evalúa si vale la pena seguir adelante con la solución S actual. Esto es, se evalúa si dada la S actual y todos los elementos que quedan por meter, si se metiesen todos los elementos restantes en alguna máquina, se alcanzaría un beneficio mayor al que se tiene computado como el máximo hasta ahora.

En caso de que se determine que vale la pena seguir con la S actual, se busca el máximo beneficio obtenible al agregar a la solución S actual el elemento m (es decir, al asignar la siguiente instancia a la máquina m) de la misma forma que en el algoritmo de fuerza bruta y se lo retorna. Si se determina que no vale la pena, se retorna 0.

Pseudo-código:

```
llenar_maquinas_BT_opt (beneficios, S, maximo, best_solution):
    if (|S|  == |beneficios|):
        if(S es factible):
            if (beneficio de S > beneficio de best_solution):
                best_solution = s
                maximo = suma_beneficios(s)
            end if
            return maximo

        end if
        else:
            return 0
        end else
    end if
    else:
        valores = []
        if (vale_la_pena(S)):
            for (m perteneciente a {0, 1, …, n}):
                S = S U {m}
                beneficio = llenar_maquinas_FB(beneficios, S, maximo
                                                        best_solution)
                agregar beneficio a valores
            end for
        end if
        return maximo(valores)
    end else
```

Backtracking - Factibilidad

Este algoritmo es recursivo y se basa en el de fuerza bruta. Como caso base, cuando la longitud de S alcanza la cantidad de instancias, si el beneficio conseguido con esa solución es mejor que el conseguido con la mejor solución almacenada (best_solution), se almacena la nueva mejor solución y se retorna el beneficio de la solución. Si no, se retorna 0. En este caso base, no se analiza si la solución S es factible como en los dos algoritmos anteriores.

Si la longitud de S es menor a la cantidad de instancias (es decir, queden instancias por asignar), para cada m perteneciente a {0, 1, …, n} se busca el máximo beneficio obtenible luego de agregar a la solución S el elemento m si y sólo si m es 0 (en este caso no se mete la instancia en la máquina) o se determina que dicha instancia cabe en la máquina m.

Para esto, se evalúa si, dada la S actual, la suma de todos los pesos de las instancias ya asignadas a la máquina m más el peso de la siguiente instancia superan la capacidad de GPUs de la máquina m. Finalmente, se retorna el máximo beneficio obtenible.

Pseudo-código:

```
llenar_maquinas_BT_fact (beneficios, S, best_solution):
    if (|S|  == |beneficios|):
        if (beneficio de S > beneficio de best_solution):
            best_solution = S
        end if
        return beneficio de S
    end if
    else:
        valores = []
        for (m perteneciente a {0, 1, …, n}):
            if (la instancia |S| cabe en la maquina m):
                S = S U {m}
                beneficio = llenar_maquinas_FB(beneficios, S, best_solution)
                agregar beneficio a valores
            end if
        end for
        return maximo(valores)
    end else
```

Programación dinámica:

El algoritmo de programación dinámica desarrollado utiliza un enfoque top-down, es decir, parte de un caso complejo y recursivamente se acerca a un caso base. Este algoritmo no utiliza una solución S como las que utilizaban los algoritmos anteriores, sino que opera con el ítem (instancia cliente) a asignar y las capacidades disponibles (GPUs disponibles) en las máquinas. Adicionalmente, utiliza un mapa de memoización para almacenar cada solución que se va obteniendo.

Como caso base, si la instancia a asignar es 0 (si no hay instancias a asignar), se devuelve 0. Si la instancia a asignar es mayor 0, se evalúa si ya se conoce la respuesta para el problema con dicha instancia y las capacidades de las máquinas actuales. Para esto se consulta el mapa de memorización, pasando como clave del mismo la instancia con el conjunto de capacidades actual. Si se conoce la respuesta, se la devuelve, sino se la computa.

Para computar la solución dada la instancia y las capacidades actuales, se evalúa para cada m en {0, 1, … , n} el beneficio máximo obtenible de meter la instancia actual en la máquina m (donde meter la instancia en la máquina 0 es no meterla en ninguna máquina). Para ello se suma el beneficio de la instancia (si es que m no es 0 y la capacidad de la máquina m es mayor al peso de la instancia) al resultado de llamar a la misma función recursivamente disminuyendo la instancia en 1 (para eventualmente llegar a la instancia 0) y reduciendo la capacidad de la máquina m en el peso de la instancia.

Una vez obtenido el valor máximo para todos los m posibles, se guarda dicho valor en el mapa de memoización asociado a la clave compuesta por la instancia y capacidades actuales, y se lo devuelve.

Pseudo-código:

```
llenar_maquinas_pd (instancia, capacidades, pesos, beneficios, dp):
    if instancia == 0:
        return 0
    end if
    clave = (instancia, capacidades)
    if (clave en dp):
        return dp[clave]
    end if
    maximo = 0
    for (m perteneciente a {0, 1, …, n}):
        if (m == 0)
            beneficio = llenar_maquinas_pd(instancia-1, capacidades, pesos, beneficios,
                                           dp)
        end if
        if (m != 0 y pesos[instancia-1] <= capacidades[m-1])
            capacidades' = capacidades
            capacidades'[m] = capacidades'[m] - pesos[instancia-1]
            beneficio = beneficios[instancia-1] + llenar_maquinas_pd (instancia-1,
                                           capacidades', pesos, beneficios, dp)
            maximo = max(maximo, beneficio)
        end if
    end for
    dp[clave] = maximo
    return maximo
```

Algoritmo de reconstrucción de la solución para programación dinámica

Este algoritmo reconstruye la solución asociada al máximo beneficio encontrado con el algoritmo de programación dinámica. Para ello, utiliza el mapa de memoización resultante luego de aplicar dicho algoritmo. Para reconstruir la solución se inicializa un vector que almacenará la solución reconstruída, es decir, en qué recurso o máquina se asigna cada ítem (o si no se asigna). El algoritmo sigue un bucle while que se ejecuta mientras queden ítems por considerar y aún haya capacidad en las máquinas.

En cada iteración, compara si no asignar el ítem actual produce el mismo beneficio que en el estado actual. Esto se verifica usando la tabla dp:

- Si el beneficio es el mismo, significa que el ítem no fue asignado, por lo que se agrega un 0 al vector rv, se decrementa i y se continúa.

Si el beneficio no es el mismo, el ítem sí fue asignado, por lo que se busca en qué máquina se asignó:

- Se recorren las máquinas, verificando si el ítem puede ser acomodado en el recurso actual (es decir, si el peso del ítem es menor o igual a la capacidad restante de la máquina).
- Si puede ser asignado, se actualizan las capacidades y se verifica si asignarlo en este recurso produce el mismo beneficio que el valor óptimo en la tabla dp.
- Si se cumple, significa que el ítem fue asignado a ese recurso, por lo que se actualizan las capacidades y se agrega el índice de la máquina al vector rv. Luego, se decrementa el índice del ítem a asignar y se rompe el ciclo que recorre las máquinas, avanzando al siguiente ítem.

Si el ciclo principal termina cuando quedan items pero todas las capacidades ya están ocupadas, el algoritmo sigue agregando 0 al vector rv, indicando que el resto de los ítems no fueron asignados.

Al final, invierte el vector rv para que las asignaciones queden en el orden correcto, ya que se ha construido de atrás hacia adelante, comenzando desde el último ítem. Así, el vector resultado contiene para cada ítem un número que indica a qué máquina fue asignado (o un 0 si no fue asignado a ninguno).

Pseudo-código:

```
reconstruir(dp, item, capacidades, pesos, beneficios):
    res = []
    while (items > 0 y queda espacio en alguna máquina):
        if (dp[item, capacidades)] == dp[item-1, capacidades)]):
            i = i -1
            agregar 0 a res
        end if
        else:
            for (m perteneciente a {1, … , |capacidades| }):
                if (pesos[item-1] <= capacidades[m-1]):
                    capacidades' = capacidades
                    capacidades'[m] -= pesos[instancia-1]
                    if (dp[(item, capacidades] == dp[item-1, capacidades'):
                        capacidades = capacidades'
                        agregar m a res
                        i = i -1
                        exit for
                    end if
                end if
            end for
        end else
    end while
    while (item > 0):
        agregar 0 a res
        i = i -1
    end while
    invertir res
    return res
```

Observaciones sobre los algoritmos y su implementación:

- Los algoritmos computan el beneficio máximo obtenible. Los de fuerza bruta y backtracking modifican el best_solution pasado por parámetro, donde queda almacenada la mejor solución al problema. El de programación dinámica no reconstruye la solución, sino que debe usar de forma auxiliar el algoritmo de reconstrucción descrito.
- Para que los algoritmos de fuerza bruta y backtracking puedan modificar el best_solution pasado por parámetro, el mismo se pasa por referencia en C++ y como variable global en Python.
- Para el algoritmo de programación dinámica, en C++ se usa un map como mapa de memoización, mientras que en Python se utiliza un diccionario. Mientras que los map de C++ permiten utilizar vectores como claves, Python no permite utilizar listas como claves de los diccionarios. Por este motivo, en la implementación de Python se convierte la lista capacidades a tupla cuando se la quiere utilizar como parte de la clave (item, capacidades).
- Funcionalidades tales como:
    - evaluar la factibilidad de una solución,
    - calcular el beneficio de una solución,
    - buscar el máximo de un arreglo o vector,
    - evaluar si la instancia i cabe en la máquina m,
    - evaluar si vale la pena seguir con la solución actual (en optimalidad),
    - invertir un vector,
    - evaluar si queda lugar en alguna máquina,

  fueron resueltas en funciones auxiliares.

Comparación sobre la eficiencia de los algoritmos

Comenzando por el algoritmo de fuerza bruta, el mismo posee una complejidad computacional de O(m^n), tanto en peor caso como en el caso promedio, donde m es la cantidad de máquinas disponibles y n la cantidad de ítems. Esto se debe a que el algoritmo siempre analiza todas las posibles soluciones y las soluciones están compuestas por n elementos que pueden tomar m valores.

Los algoritmos de backtracking, en peor caso, tendrán también complejidad O(n). Dicho peor caso sería que todos las instancias entren en una sola máquina para el de poda por factibilidad (la poda por factibilidad nunca se aplica), y que los ítems se presenten de forma tal que siempre cada nueva solución otorgue un beneficio mayor al máximo guardado para el de poda por optimalidad. Sin embargo, la complejidad debería reducirse para el caso promedio. Esto será estudiado luego en la sección de experimentación.

Finalmente, el algoritmo de programación dinámica tendrá una complejidad en peor caso de O(n*m*c^m]) donde c es la máxima capacidad de una máquina. El de reconstrucción de la solución, por su parte, tendrá una complejidad en peor caso de O(n* m*|dp|) o O(n*m*log(|dp|) dependiendo en qué lenguaje y con qué tipo de estructura de datos se implemente el dp. Así, la combinación de hallar el beneficio máximo y reconstruir la solución será la suma de las dos complejidades anteriores. En un principio, es difícil determinar cómo se comportará este algoritmo con respecto a los tres anteriores.

Experimentación, resultados y discusión

Se realizó una serie de experimentos para comprender mejor el funcionamiento y comportamiento de los distintos algoritmos desarrollados al variar distintos parámetros.

La metodología general para todos fue la siguiente:

- Hardware: Intel Core i7-10700K, 16 GB RAM DDR4, SSD 512GB.
- Sistema operativo: Windows 10
- Software: C++ con g++ 11.3.0.
- Diseño experimental: se corrió cada algoritmo 10 veces para promediar el tiempo de ejecución de cada uno

Experimento I: Cada máquina puede albergar la totalidad de las instancias

Metodología particular:

- Datos utilizados: se trabajó perímetro con 5 instancias y luego 10. Los pesos de las mismas se eligieron manualmente de forma tal que la sumatoria de los mismos sea menor o igual a la capacidad de cada una de las mochilas. Los beneficios se generaron siguiendo una distribución normal de media 250 y desvío estándar 100. Se definen 2 máquinas, ambas con capacidad 1000.

  Beneficios_5: {393, 335, 242, 348, 269}
  Pesos_5: {150,300,200,100,200}

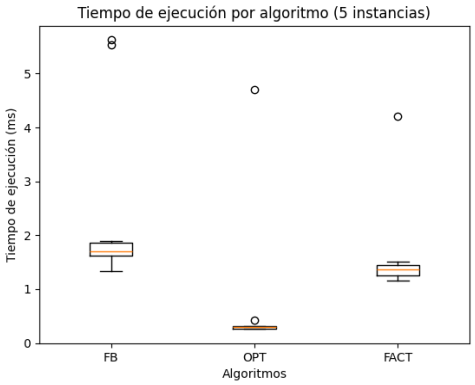
  Beneficios_10: {224, 301, 184, 452, 164, 304, 19, 118, 346, 288}
  Pesos_10: {50,70,10,60,80,50,50,70,10,60}

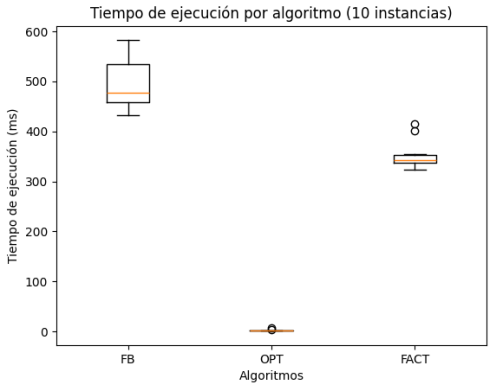
Hipótesis:

Al entrar todas las instancias en una sola de las máquinas, sea cual sea, la poda por factibilidad debería ser poco eficiente. Por ello, se espera que el algoritmo de fuerza bruta y poda por factibilidad tengan rendimientos similares, siendo los peores en desempeñarse. Además, se espera ver que con la distribución normal de los beneficios la poda por optimalidad mejore considerablemente el rendimiento de este algoritmo por sobre los dos anteriores.

Resultados:

Los resultados obtenidos se presentan en los siguientes boxplot:





Como se esperaba, tanto para 5 como para 10 instancias, los algoritmos de fuerza bruta y poda por factibilidad se comportaron similar, ambos peor que el de optimalidad, que dado a la normalidad de los beneficios tuvo una poda medianamente efectiva que mejoró su complejidad algorítmica promedio. Sin embargo, se observa una leve diferencia entre FB y FACT. Se ejecutó el experimento varias veces, y dicha diferencia siempre estuvo presente. Por ello, descartamos la posibilidad de que sea una casualidad.

Teorizamos que la diferencia observada en el desempeño de FB y FACT se deben a la presencia de la función pesos_ok() en el algoritmo de FB. Esta función puede ser la que esté generando un costo extra, que se ahorra para FACT al precomputar la factibilidad de una solución en cada paso sin uso de mapas.

<u>Experimento II: Beneficios ascendentes</u>

Metodología particular:

- Datos utilizados: se trabajó con 10 instancias. Los pesos de las mismas se generaron con una distribución normal de media 500 y desvío estándar 250. Los beneficios se generaron manualmente de forma tal que sean ascendentes. Se definen 2 máquinas, ambas con capacidad 1000.

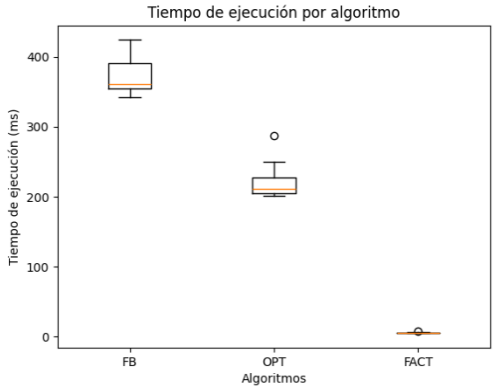
  Beneficios: {98, 221, 347, 459, 581, 718, 903, 1059, 1192, 1198}
  Pesos: {918, 591, 735, 592, 76, 683, 704, 363, 262, 492}

Hipótesis:

Al estar los beneficios ordenados de forma ascendente, cada nueva solución que el algoritmo con poda por optimalidad encuentre debería ser mejor que la anterior, es decir debería reducirse la cantidad de veces que una solución se aborta por no “valer la pena”, bajando el rendimiento de dicha poda. Por ello, se espera que el algoritmo de fuerza bruta y poda por optimalidad tengan rendimientos similares, siendo los peores en desempeñarse. Además, se espera ver que con la distribución normal de los pesos, la poda por factibilidad mejore considerablemente el rendimiento de este algoritmo por sobre los dos anteriores.

Resultados:

Los resultados obtenidos se presentan en el siguiente boxplot:



Como se esperaba, el algoritmo de FACT vio una mejora en su rendimiento, siendo el algoritmo que mejor se desempeñó. Sin embargo, no se cumplió que OPT y FB se comporten de manera tan similar. Si bien el ordenar los beneficios de forma ascendente empeoró el rendimiento de OPT, no lo llevó a su complejidad de peor caso como esperábamos.

Experimento III: Ninguna instancia se puede asignar

Metodología particular:

- Datos utilizados: se trabajó con 10 instancias. Los pesos de las mismas se definieron todos como un mismo valor mayor al máximo soportado por las máquinas. Los beneficios se generaron con distribución normal con media 250 y desvío estándar 100. Se definen 2 máquinas, ambas con capacidad 1000.

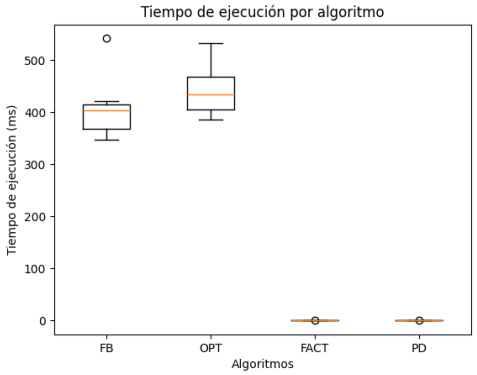
  Beneficios: {224, 301, 184, 452, 164, 304, 19, 118, 346, 288}
  Pesos: {1001, 1001, 1001, 1001, 1001, 1001, 1001, 1001, 1001, 1001}

Hipótesis:

Como ninguna instancia entra en ninguna máquina, se espera que los algoritmos de factibilidad y programación dinámica sean los que mejor se desempeñen. El primero tendrá una alta tasa de poda de soluciones no factibles. El segundo, debería llegar a los casos base del mapa de memoización. Además, se espera ver que con la distribución normal de los beneficios, la poda por optimalidad mejore considerablemente el rendimiento de este algoritmo por sobre el de fuerza bruta.

Resultados:

Los resultados obtenidos se presentan en el siguiente boxplot:



Como se esperaba, FACT y PD se desempeñaron mejor que sus competidores. Sin embargo, no se observó que OPT se desempeñe mejor que FB. Esto es extraño y se contradice con el comportamiento observado en el Experimento I, donde para 10 instancias el algoritmo OPT tardaba en promedio unos 20 ms.

Como la única variable que varió para ambos experimentos son los pesos, teorizamos que los mismos tienen una incidencia sobre el desempeño del algoritmo de OPT mayor al que esperábamos y que se debería seguir estudiando.

Experimento IV: Pesos bajos vs altos

Metodología particular:

- Datos utilizados: se trabajó con 10 instancias. Se generaron dos vectores de pesos con distribución normal, uno de media 250 y desvío estándar 100, y el otro con media 600 y desvío estándar 100. Los beneficios también se generaron con distribución normal con media 500 y desvío estándar 150. Se definen 2 máquinas, ambas con capacidad 1000.

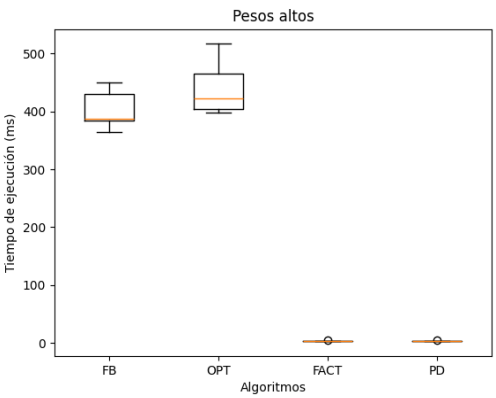
  Beneficios: {673, 641, 459, 789, 551, 746, 522, 271, 525, 448}
  Pesos_bajos: {76, 200, 345, 226, 269, 244, 144, 278, 363, 296}
  Pesos_altos: {554, 602, 503, 627, 464, 546, 603, 282, 658, 705}

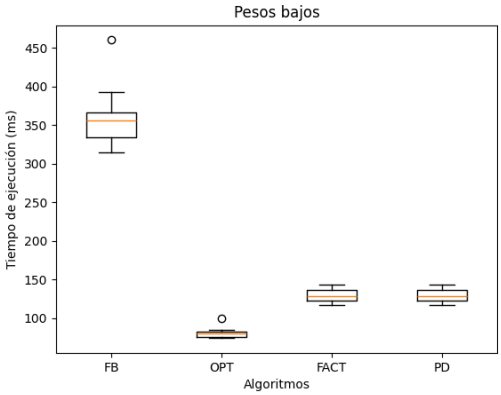
Hipótesis:

Se desea ver que para los pesos bajos, los algoritmos de poda por factibilidad, optimalidad y programación dinámica se comportan similar, todos mejor que el de fuerza bruta, mientras que para los pesos altos, el de programación dinámica y el de factibilidad ganan en eficiencia, superando al de optimalidad.

Resultados:

Los resultados obtenidos se presentan en los siguientes boxplot:





Como se esperaba, los algoritmos FACT y PD aumentaron su rendimiento al aumentar los pesos. Sin embargo, también se encontró que OPT empeoró notablemente su rendimiento, incluso superando a fuerza bruta. Esto, reafirma lo teorizado luego del experimento III; que los pesos tienen una incidencia sobre el desempeño de OPT que no tenemos en cuenta. Lo que puede estar sucediendo es que las funciones auxiliares utilizadas en este algoritmo como pesos_ok() y vale_la_pena() estén elevando la complejidad más de lo esperado.

Experimento V: 4 máquinas chicas vs. dos grandes

Metodología particular:

- Se comparó la performance de los distintos algoritmos tanto para pesos bajos como pesos altos para dos opciones distintas, 4 máquinas de 500 GPUs cada una, o dos de 1000.
- Adicionalmente se compara el máximo beneficio obtenible para cada cantidad de máquinas.
- Datos utilizados: Los pesos (altos y bajos) y beneficios se generaron igual que en el experimento IV, pero para 5 instancias

Beneficios: {301, 797, 479, 525, 490}
Pesos_bajos: {76, 200, 345, 226, 269}
Pesos_altos: 548, 423, 279, 744, 442}

Observación: se realizó el experimento con 5 instancias dado que para 10 instancias y 4 máquinas los algoritmos de fuerza bruta y optimalidad tardaban una elevada cantidad de tiempo.

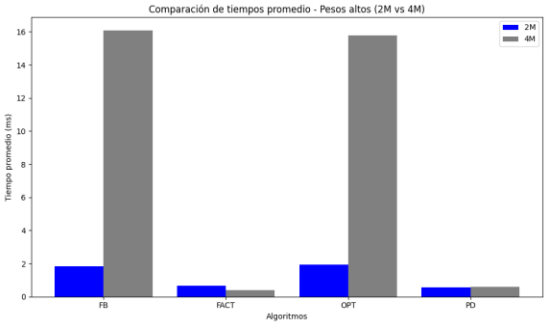
Hipótesis:

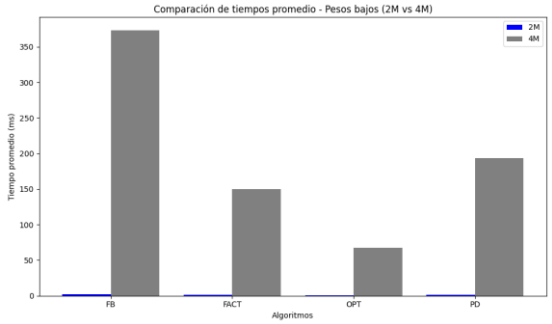
Para 4 máquinas de 500 GPUs se observará un mejor rendimiento de los algoritmos de programación dinámica y poda por factibilidad que para 2 máquinas de 1000 GPUs, tanto para los pesos altos y bajos. Esto se debe a que al cada máquina tener menor capacidad, cada una se llena más rápido, por lo que estos dos algoritmos ganarán eficiencia.

En cuanto a los beneficios obtenibles, se espera que para pesos bajos, el beneficio máximo obtenible para cada cantidad de máquinas no varíe mucho. Para pesos altos, en cambio, se espera que si haya una mayor diferencia entre lo obtenible con 2 o 4 máquinas.

Resultados:

Los resultados en cuanto a desempeño algorítmico se muestran en los siguientes gráficos:





Para pesos bajos, si bien con 2 máquinas todos los algoritmos se comportan de manera muy similar, para 4 máquinas no se cumple lo esperado en la hipótesis. Aunque FACT y PD se comportan de manera similar, son superados por OPT. Además parece que el patrón encontrado de OPT en los experimentos III y IV se mantiene tanto para 4 máquinas como para 2; para pesos bajos se comporta eficientemente, pero para pesos se demora igual o incluso más que FB.

Para pesos altos, si se cumple la planteado en la hipótesis; tanto para 4 máquinas como para 2, FACT y PD son los que tienen mejor desempeño.

Los resultados en cuanto a beneficio máximo se muestran a continuación:

		maquinas	
		2 de 1000	4 de 500
pesos	altos	2067	1766
	bajos	2592	2592

Los resultados indican evidencia a favor de la hipótesis.

Experimento VI: tiempo de ejecución variando tamaño de entrada, cantidad de máquinas y pesos

Metodología particular:

- Se comparó la performance de los distintos algoritmos variando la cantidad de instancias (n a partir de ahora), la cantidad de máquinas, y los pesos. Se comparó por un lado la tardanza de cada algoritmo en función de n para 2 máquinas de 1000 GPUs y 4 de 500 GPUs con pesos bajos. Por otro lado, se comparó la tardanza de cada algoritmo en función de n para pesos altos y pesos bajos en 2 máquinas de 1000 GPUs.
- Datos utilizados: Los pesos (altos y bajos) y beneficios se generaron igual que en el experimento IV, y para variar la cantidad de instancias a probar se seleccionó un recorte del vector de pesos y beneficios de 10 elementos.

  Beneficios: {98, 221, 347, 459, 581, 718, 903, 98, 221, 347}
  Pesos_bajos: {107, 226, 565, 195, 216, 315, 252,565, 195, 216}
  Pesos_altos: {525, 565, 736, 644, 470, 152, 430, 363, 778, 794}

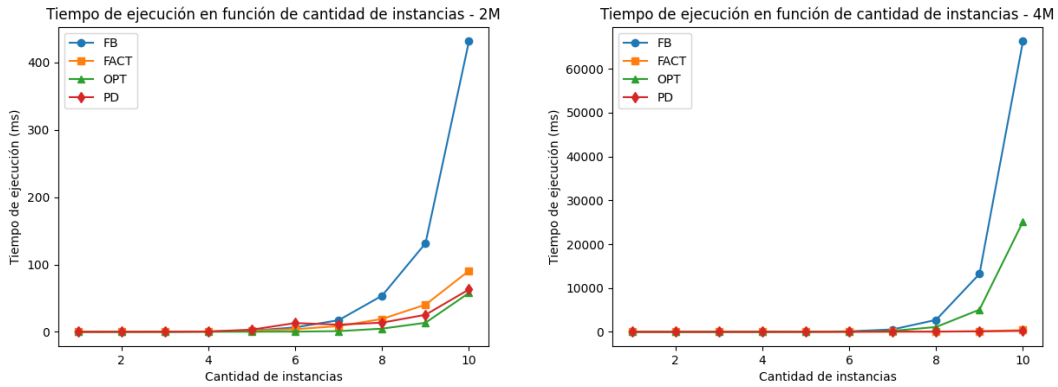
Hipótesis:

En la comparación por cantidad de máquinas y pesos bajos, para dos máquinas FACT, OPT y PD se comportarán similarmente, mientras que para 4 máquinas, OPT se volverá menos eficiente.

Según lo visto en los experimentos III, IV y V, para la comparación por pesos para 2 máquinas se espera que para pesos altos, la diferencia entre optimalidad y fuerza bruta sea mayor en pesos bajos que en pesos altos.

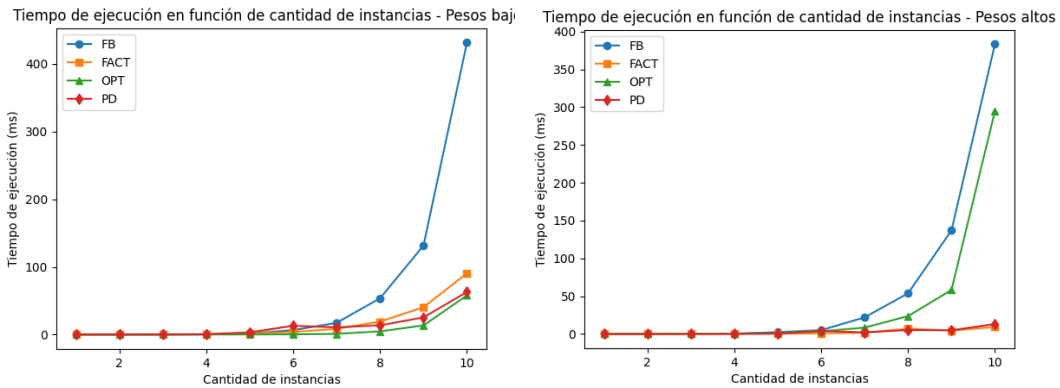
Resultados:

Comparación por cantidad de máquinas con pesos bajos:



Como se puede ver en las gráficas la hipótesis propuesta era acertada.

Comparación por pesos con 2 máquinas de 1000 GPUs:



Las gráficas muestran evidencia a favor de la hipótesis propuesta.

Experimento VII: Python vs. C++

Metodología particular:

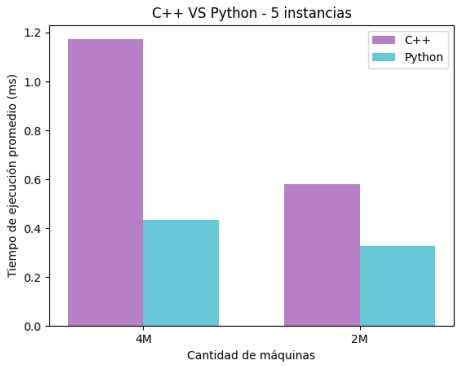
- Se comparó la performance del algoritmo de programación dinámica variando el lenguaje en el que se lo implementa, la cantidad de máquinas (2 de 1000 GPUs o 4 de 500) y la cantidad de instancias.
- Software: C++ con g++ 11.3.0, Python 3.10.11.
- Datos utilizados: Se emplearon pesos y beneficios normales con media 500 y desvío estándar 250

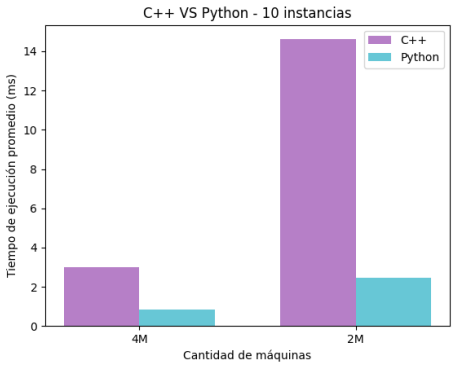
  Pesos_5: {548, 423, 279, 744, 442}
  Pesos_10: {673, 989, 588, 658, 628, 630, 369, 370, 785, 220}
  Pesos_20: {954, 314, 358, 420, 609, 305, 598, 420, 292, 153, 543, 720, 592, 372, 1002, 619, 162, 673, 415, 812}
  Beneficios_5: {98, 221, 347, 459, 581}
  Beneficios_10: {617, 647, 362, 435, 243, 537, 173, 559, 428, 396}
  Beneficios_20: {486, 535, 226, 454, 384, 741, 595, 956, 529, 713, 268, 223, 410, 335, 191, 328, 20, 322, 711, 727}

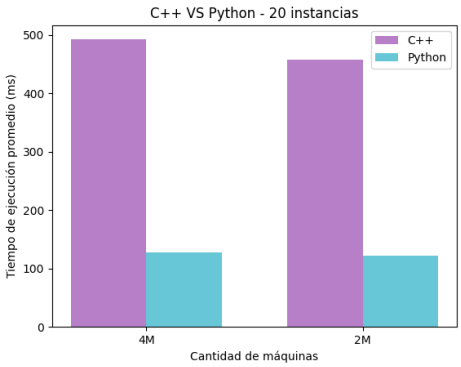
Hipótesis:

Dado que C++ es un lenguaje compilado, lo que significa que el código se traduce directamente a código máquina antes de la ejecución, y Python es un lenguaje interpretado, lo que implica que el código se traduce a código máquina y genera una sobrecarga adicional en cada ejecución, para toda cantidad de máquinas y de instancias, la implementación en C++ será más rápida que la de Python.

Resultados:







De manera completamente opuesta a lo esperado, las implementaciones en Python fueron las más rápidas en todos los casos. Teorizamos que esto puede deberse a la implementación interna que tenga cada lenguaje de los mapas/diccionarios. A futuro, se puede experimentar cambiando la implementación de C++ para que funcione con unorderd_map´s en lugar de map´s. Esto debería mejorar los tiempos de acceso a memoria en caso promedio.

Conclusiones

Este trabajo investigó diversas estrategias para resolver el problema de asignación óptima de GPUs en la nube, implementando y evaluando cuatro algoritmos: fuerza bruta, backtracking con poda por optimalidad y factibilidad, y programación dinámica. A través de experimentos comparativos, se identificó que los algoritmos de programación dinámica y poda por factibilidad ofrecieron el mejor rendimiento en la mayoría de los escenarios, mientras que el algoritmo de fuerza bruta presentó las peores prestaciones, confirmando la hipótesis sobre su ineficiencia en casos más complejos.

Uno de los principales hallazgos es que, en escenarios con beneficios o pesos estructurados (ascendentes o muy elevados), los algoritmos de programación dinámica y factibilidad no solo lograron mejores resultados en cuanto a eficiencia de tiempo, sino que también permitieron una mejor utilización de los recursos. A su vez, cuando las instancias a asignar tienen pesos elevados, el algoritmo de poda por optimalidad no se comporta tan eficientemente como se anticipaba, incluso superando en tiempo de ejecución al de fuerza bruta en algunos casos. Esto indica que la estructura de los pesos tiene una mayor influencia sobre su rendimiento de lo que inicialmente se estimaba, lo que sugiere que es necesario seguir investigando este fenómeno para comprender mejor su impacto en la eficiencia del algoritmo.

En función de los resultados obtenidos, se recomienda la adopción de algoritmos de programación dinámica para resolver problemas similares, debido a su balance entre complejidad y tiempo de ejecución. Sin embargo, en casos donde los beneficios o pesos sean altamente distribuidos, los algoritmos de poda por factibilidad también resultan una opción eficiente.

En cuanto a recomendaciones para los servidores, la opción de utilizar 4 máquinas con 500 GPUs es menos conveniente que la de 2 máquinas de 1000 GPUs. Si bien en casos donde las instancias son numerosas o cuando los pesos de las mismas son elevados los algoritmos para asignar recursos se ejecutan con mayor eficiencia para 4 máquinas, el beneficio máximo obtenible es menor, dado que cada máquina se llena con mayor facilidad.

Las limitaciones del estudio incluyen la escala relativamente pequeña de los experimentos en cuanto al número de instancias y máquinas disponibles. Además, no se evaluaron otros factores, como variaciones en la carga de trabajo de las GPUs o en la distribución temporal de las solicitudes de los clientes. Para investigaciones futuras, se sugiere ampliar el rango de instancias y explorar enfoques híbridos que combinen las mejores características de los algoritmos evaluados, además de mejorar la implementación en C++ para optimizar el acceso a memoria. Esto podría reducir la sorprendente ventaja observada en la implementación en Python.