Informe TP1
Algoritmos exactos para el KP-01 con Grafo de Conflictos

Integrantes: Josefina Casas Pardo, Paula Ho, Denis Wu

---

1. Introducción

En el siguiente informe, nos proponemos a abarcar y modelar el *Problema de la Mochila 0-1 con Grafo de Conflictos* (KP01wCG). Se trata de una extensión del famoso problema KP01, para el que se imponen restricciones de incompatibilidad entre ítems, impidiendo la selección simultánea de ciertos pares.

Así, el presente modelo resulta relevante en aplicaciones como la optimización de carteras financieras o selección de proyectos, donde distintas regulaciones pueden prohibir la simultaneidad de ciertos activos. De igual manera, a la hora de asignar procesos a procesadores, puede darse el caso de que dos procesos sean incapaces de destinarse a un mismo procesador al competir por un mismo recurso, como una memoria compartida [1].

El objetivo del trabajo consiste en diseñar distintos algoritmos para el problema de KP01wCG, comparando luego la eficiencia computacional según la implementación y lenguaje de programación utilizado.

El informe estará dividido en siete partes. En *Descripción del problema*, detallaremos de modo claro y preciso el problema KP01wCG. En *Modelado*, elaboraremos una representación descriptiva y gráfica de la solución, y después en *Algoritmos implementados*, explicaremos cuáles fueron los algoritmos seleccionados y cómo fueron implementados. En *Hipótesis*, plantearemos y desarrollaremos las hipótesis del trabajo, y en *Experimentación*, realizaremos algunos experimentos para respaldar y validar las mismas hipótesis, discutiendo sobre los datos obtenidos en *Resultados y discusión*. Finalmente en *Conclusión*, revisaremos los hallazgos del trabajo y de qué manera responden a las hipótesis formuladas.

2. Descripción del problema

Los problemas de optimización combinatoria son aquellos en los que se busca encontrar una selección óptima dentro de un conjunto de elementos, que suele ser usualmente muy grande. Particularmente, en el *Problema de la Mochila 0-1* (KP01), dado un conjunto de ítems $N = \{1, \ldots, n\}$, donde cada ítem $i \in N$ tiene un beneficio $p_i > 0$ y un peso $w_i > 0$, el objetivo es seleccionar un subconjunto tal que se maximice el beneficio total, sin que el peso total supere la capacidad límite de la mochila definida como C.

En el *Problema de la Mochila 0-1 con Grafo de Conflictos* (KP01wCG), se añade una nueva restricción, en la que algunos ítems no pueden seleccionarse simultáneamente para la mochila. Estas incompatibilidades se representan mediante un *grafo de conflictos* $G = (N,E)$, donde cada arista $(i, j) \in E$ indica un conflicto entre los ítems i y j, de modo que no pueden ser seleccionados al mismo tiempo.

De esta manera, el objetivo es determinar un subconjunto de ítems $S \subseteq N$, tal que:

- $\sum_{i \in S} w_i \leq C$ (restricción de capacidad).
- No existen $i,j \in S$ tales que $(i,j) \in E$ (restricción de conflicto).
- El beneficio total $\sum_{i \in S} p_i$ sea máximo (solución óptima).

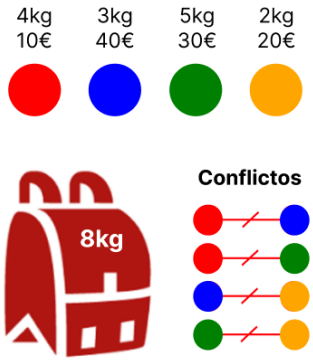

Fig. 1 Ejemplo ilustrativo del problema KP01wCG

3. Modelado

Dada una instancia con N ítems, un primer modelado del problema consiste en explorar el conjunto de soluciones formado por todas las posibles combinaciones de selección de los ítems de la instancia.

Así, una representación posible es mediante un grafo de árbol, cuya raíz parte de un conjunto de solución vacío. La exploración de ítems sigue una estrategia de ramificación, y en cada nivel del árbol, el conjunto se divide en dos posibles caminos: incluir o no el ítem actual.

Considerando este enfoque, la cantidad de combinaciones posibles de ítems es de 2^N, creciendo exponencialmente a medida que aumenta N. Por tal motivo, uno de los objetivos del trabajo sería incluir podas con las que se eliminen ciertas ramificaciones, de tal modo que se encuentre la solución óptima sin la necesidad de construir un árbol completo.

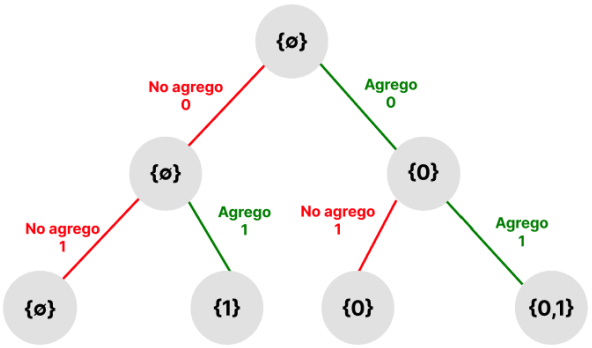

Fig. 2 Modelo de la solución por grafo de árbol

Además del grafo de árbol, utilizaremos un segundo modelo que tiene como motivación el enfoque de programación dinámica.

Dada una instancia con n ítems y una capacidad máxima C, el modelo se representa como una tabla, donde cada celda indica el beneficio máximo alcanzable con una determinada cantidad de ítems y capacidad (remanente).

Así, la tabla se estructura como una matriz de (n+1)x(C+1), donde cada fila k simboliza las decisiones realizadas sobre los primeros k ítems, y cada columna c evalúa cuál es el beneficio máximo que puede alcanzarse sin exceder la capacidad remanente.

4. Algoritmos implementados

El primer algoritmo que abordamos es una estrategia simple basada en fuerza bruta. De manera exhaustiva, este enfoque recorre recursivamente todas las combinaciones posibles de subconjuntos de ítems. Así, a medida que avanza el algoritmo, se actualiza la mejor solución descubierta cada vez que se encuentra otra solución factible cuyo beneficio total sea superior. Finalmente, se obtendría la solución óptima una vez que se haya realizado el recorrido completo. En *Algoritmo 1*, se muestra una estructura del algoritmo de BruteForceKP01wCG.

Algoritmo 1: BruteForceKP01wCG

Input: BruteForceKP01wCG instance
Output: Solución óptima

1. (Inicialización) Sean una solución parcial vacía S y un ítem k inicializado en 0.
2. (División) Generar dos soluciones parciales S1 y S2 incluyendo o no el ítem k respectivamente.
3. (Recorrido completo) Si k = n, significa que se ha recorrido completamente la instancia. En tal caso, avanzar al paso 4. En caso contrario, volver al paso 1 por separado con S1 o S2 y k = k + 1.
4. (Evaluación) Verificar si la solución S es factible, es decir, si el peso total es ≤ a la capacidad de la mochila, y si los ítems no presentan conflicto entre sí. Si es el caso y además su beneficio total supera al de la mejor solución encontrada hasta el momento, actualizar la mejor solución.
5. (Terminación) Devolver la mejor solución una vez que todas las combinaciones posibles hayan sido evaluadas.

El segundo algoritmo que implementamos es una mejora directa de la estrategia de fuerza bruta. La idea sería incluir podas de factibilidad y optimalidad en los casos recursivos, con el fin de reducir el número de combinaciones de ítems a evaluar.

Particularmente en este problema, definimos como podas de factibilidad descartar soluciones cuyo peso parcial exceda la capacidad de la mochila, y eliminar ramas donde el ítem a incluir presenta conflicto con algunos de los ítems ya seleccionados. Luego, la poda por optimalidad excluye soluciones parciales donde aún considerando el beneficio de los ítems restantes, el beneficio total no

supera al de la mejor solución conocida. En *Algoritmo 2*, se muestra una estructura del algoritmo de BackTrackingKP01wCG.

Algoritmo 2: BackTrackingKP01wCG

Input: BacktrackingKP01wCG instance
Output: Solución óptima

1. (Inicialización) Sean una solución parcial vacía S y un ítem k inicializado en 0.
2. (Podas) Si el peso total supera a la capacidad límite C (factibilidad 1), o el ítem k entra en conflicto con algún ítem ya seleccionado en S (factibilidad 2), o bien la suma del beneficio total considerando los beneficios restantes es menor al beneficio de la mejor solución (optimalidad), volver al paso 1 con k = k + 1 sin agregar el ítem k. En caso contrario, avanzar al paso 3.
3. (División) Generar dos soluciones parciales S1 y S2 incluyendo o no el ítem k respectivamente.
4. (Recorrido completo) Si k = n, significa que se ha recorrido completamente la instancia. En tal caso, avanzar al paso 5. En caso contrario, volver al paso 1 por separado con S1 o S2 y k = k + 1.
5. (Evaluación) Verificar si la solución S es factible, es decir, si el peso total es ≤ a la capacidad de la mochila. Si es el caso y además su beneficio total supera al de la mejor solución encontrada hasta el momento, actualizar la mejor solución. (Tener en cuenta que en BT no haría falta verificar los conflictos entre ítems en el caso base, pues lo mismo ya fue considerada en los casos recursivos).
6. (Terminación) Devolver la mejor solución una vez que todas las combinaciones posibles hayan sido evaluadas.

La tercera estrategia se basa en la programación dinámica, para la cual ignoraremos las restricciones de conflicto entre ítems, refiriendo al problema clásico de KP01. La motivación de este algoritmo consiste en evitar cálculos redundantes mediante el almacenamiento y reutilización de soluciones previamente computadas.

Particularmente, para implementar este algoritmo utilizamos el enfoque bottom-up. La idea se basa en construir una tabla de beneficios de manera iterativa, modelando la decisión de incluir o no cada ítem en función del beneficio que aporta y la capacidad restante de la mochila. En *Algoritmo 3*, se muestra una estructura del algoritmo de DynamicProgrammingKP01.

Algoritmo 3: DynamicProgrammingKP01 (enfoque bottom-up)

Input: DynamicProgrammingKP01 instance
Output: Solución óptima

1. (Inicialización) Sean una capacidad *C* y una cantidad de ítems *n*.
2. (Construcción) Construir la tabla como una matriz *m* de (n+1)x(C+1). Inicializar en 0 la primera fila y la primera columna dado que son casos vacíos.
3. (Rellenado) Completar cada fila de la tabla definiendo si incluir el ítem correspondiente *k* dada una

capacidad remanente *c* ($1 \leq c \leq C$). Así, si el peso de *k* supera a *c*, m[k][c] = m[k - 1][c]. En caso contrario, comparar el valor de beneficio entre incluir y no el ítem *k*, de modo que m[k][c] sea el máximo entre m[k - 1][c] y b[k] + m[k - 1][c - p[k]].

4. (Reconstrucción) Una vez que la tabla está llena, reconstruir la solución partiendo desde el último valor de la matriz hasta el origen (k = 0 ó c = 0). En cada paso, si m[k][c] = b[k] + m[k - 1][c - p[k]], significa que se debe añadir el ítem *k* en la solución. En caso contrario, no se incluye el ítem y se vuelve a verificar lo mismo con el *k* - 1.

4.1. Estructuras de datos

Para las implementaciones en C++, construimos tres estructuras de datos principales: Graph, KP01withCGInstance y Solution.

En primer lugar, el grafo de conflictos se representa por Graph, que internamente contiene una matriz de adyacencia, donde cada posicion (i,j) muestra si existe incompatibilidad entre dichos ítems.

Luego, la instancia de ítems se modela mediante la clase KP01withCGInstance. De esta manera, los ítems se almacenan en un vector<tuple<int, int>>, donde cada tupla indica el peso y el beneficio del i-ésimo ítem.

Finalmente, la clase Solution es una abstracción que simula una solución parcial o completa del problema. Contiene un vector<int> que representa los ítems seleccionados, y un mapa para asociar cada ítem con su determinado peso y beneficio.

En la siguiente tabla, se detalla para cada instancia los métodos invocables con su correspondiente complejidad temporal.

Class	**Métodos**	**Complejidad**
KP01withCGInstance	int getWeight(index)	O(1)
	int getProfit(index)	O(1)
	int getNumItems()	O(1)
	int getCapacity()	O(1)
	int getWeightTotal()	O(1)
	int getProfitTotal()	O(1)
	void cargarDatos(archivo)	$O(N^2+C)$
	void setWeightProfit(index, weight, profit)	O(1)
	int cantidadConflictos()	O(1)
	bool hasConflict(solution, item)	O(S)
	bool hasConflictTotal(solution)	$O(S^2)$
Graph	void construirMatriz(cantidad_items)	$O(N^2)$
	vector<vector<bool>> getMatriz()	O(1)
	void addConflict(item1, item2)	O(1)
	void removeConflict(item1, item2)	O(1)
	int getCantidadConflictos()	O(1)

Solution	void addItem(item, peso, beneficio)	O(log(N))
	void removeItem(item)	O(N)
	int getWeightSolution()	O(1)
	int getProfitSolution()	O(1)
	vector<int> getItems()	O(1)
	bool contains(item)	O(log(N))

Para implementar los algoritmos en Python, recurrimos a la ayuda de ChatGPT. En el Prompt, describimos un breve contexto del problema, y le solicitamos al asistente IA los códigos para los algoritmos de BTKP01wCG y DPKP01.

Link ChatGPT: https://chatgpt.com/share/67f1cd4a-1ec8-800f-a27d-c8f4471ce740

En cuanto a las estructuras de datos, notamos que es similar a las propuestas en C++, teniendo clases como MatrizConflictos, InstanciaMochila y SolucionMochila. Un pequeño cambio es que en lugar de crear un mapa para asociar cada ítem con su peso y beneficio, en Python se utiliza una clase Item que simula el objeto con su índice, peso y beneficio correspondiente. Esto ciertamente disminuye la complejidad temporal del método addItem() en O(1), y aunque en un principio queríamos implementar la clase Item dentro de C++ tras el descubrimiento, por cuestiones de tiempo descartamos la opción. De todos modos, sería interesante observar si esto afecta de alguna manera la comparación de la eficiencia computacional entre los dos lenguajes.

Por otro lado, las implementaciones realizadas en C++ y Python difieren en su enfoque de diseño. Mientras que en C++ se prioriza la eficiencia mediante estructuras compactas como vectores de tuplas para los ítems, optimizando el acceso en memoria y velocidad de ejecución, Python opta por un diseño más legible usando clases dedicadas (Item, SolucionMochila, sacrificando rendimiento por claridad en el código. Además, algo que diferencia Python de C++ es el hecho de que los atributos no son privados y pueden ser accedidos directamente, reforzando todavía más la facilidad que tiene el lenguaje en la implementación.

4.2. Complejidad de los algoritmos

En primer lugar, dado el algoritmo de Fuerza Bruta, como en cada nivel de decisión se consideran dos alternativas (incluir o no el ítem actual), el total de combinaciones posibles es de 2^N, tal como mencionamos anteriormente. Luego, como en cada caso base se evalúa la factibilidad de la solución mediante el método hasConflictTotal() en $O(S^2)$, y en cada caso recursivo se agrega y elimina un ítem en O(log(N) + N), concluimos en que la complejidad total del algoritmo es de $O(2^N. (S^2 + N))$. Como aclaración, acá se verifica que para el caso de Python, la complejidad teórica se reduce puesto que agregar y eliminar el ítem tiene un costo de O(1 + N).

Para el caso de BackTracking, las condiciones de podas permiten evitar una parte significativa del espacio de búsqueda. Si bien teóricamente, el algoritmo sigue siendo de complejidad exponencial, puesto que se continúa dividiendo en cada nivel en dos alternativas, la complejidad práctica mejora

notablemente a medida que las podas sean lo **suficientemente** eficientes. Incluso, como el chequeo de conflictos se realiza en los casos recursivos por hasConflict() en O(S), además de que no haría falta verificar más conflictos en los casos base, la complejidad del método reduce significativamente de una cuadrática a una lineal. Así, el algoritmo de BackTracking tiene un costo de $O(2^N. (S + N))$.

Finalmente, el algoritmo de Programación Dinámica (bottom-up) evita completamente la exploración del árbol. La complejidad teórica se reduce a una pseudo polinomial, ya que consta de llenar una matriz de (N+1)x(C+1). Así, como para completar cada celda, el algoritmo únicamente realiza operaciones constantes, el costo de rellenado es de O(N.C). Luego, para reconstruir la solución, se recorre por fila desde la última celda. Nuevamente, mientras que en C++ se ejecuta en O(N. log(N)) al estar añadiendo ítems en un mapa, en Python se realiza en O(N). Por esto, la complejidad es de O(N.C + N. log(N)) para la implementación en C++, y O(N.C + N) en Python.

5. Hipótesis

Volvamos al objetivo del trabajo: comparar la eficiencia computacional en la implementación de los distintos algoritmos y lenguajes propuestos. En particular, buscamos responder las siguientes preguntas:

- ¿Hasta qué tamaño de instancias podemos abordar para el KP01wCG, y en qué tiempos de cómputo?
- ¿Cómo afecta el lenguaje de programación en la performance del algoritmo?
- ¿Cómo afecta la implementación en la performance del algoritmo?
- Tomando como problema el KP01: ¿Cómo se compara la performance de un algoritmo de backtracking con uno basado en programación dinámica?

Para las tres últimas preguntas, planteamos las siguientes hipótesis:

1. Los algoritmos implementados en C++ para KP01wCG se ejecutan en menor tiempo que los de Python, bajo las mismas condiciones de input.
2. El algoritmo de BackTracking resuelve el problema KP01wCG en menor tiempo que el de FuerzaBruta, bajo las mismas condiciones de input.
3. El algoritmo de DynamicProgramming resuelve el problema KP01 en menor tiempo que el de BackTracking, bajo las mismas condiciones de input.

La primera hipótesis se sustenta en los resultados presentados por Kumar (2019) en su artículo "*A Comparison between Python and C++*" [2]. Con el respaldo de algunos experimentos realizados, el autor concluye que, si bien Python es más amigable, más breve y más fácil de leer, C++ tiende a superar a Python en velocidad de ejecución en programas que requieren mayor procesamiento. Esto se debe a la anatomía de ambos lenguajes, pues mientras que C++ es un lenguaje compilado, donde el código se traduce una única vez y se ejecuta directamente, Python es un lenguaje interpretado cuyo código se ejecuta línea por línea, lo cual ralentiza mucho más el proceso de ejecución.

Luego, la segunda hipótesis se apoya en el comportamiento esperado de los algoritmos. Aunque ambos tienen una complejidad temporal exponencial, a diferencia de fuerza bruta, Backtracking implementa técnicas de poda que le permiten evitar la exploración completa del espacio de soluciones. Aunque en algunos casos el costo de verificar las condiciones de poda puede ser

significativo, creemos que cuando el tamaño del problema crece, dichas podas tenderían a eliminar una cantidad considerable de ramas inviables, lo que se traduce en un mejor rendimiento general en comparación con el enfoque exhaustivo de fuerza bruta.

Por último, la tercera hipótesis se basa en el análisis teórico de la complejidad de los algoritmos. Programación Dinámica, en este contexto, es un algoritmo pseudo-polinomial: su tiempo de ejecución depende linealmente de la cantidad de ítems y de la capacidad total de la mochila. Mientras tanto, el algoritmo de Backtracking tiene una complejidad exponencial. Por tal motivo, lo que esperamos sería que, cuanto más aumente el tamaño de las instancias, Programación Dinámica sea más eficiente computacionalmente.

6. Experimentación

En base a las preguntas e hipótesis formuladas, diseñamos distintos experimentos para validarlas y analizar los fenómenos observables.

Particularmente, utilizaremos los sets de archivos de instancia dados por la cátedra. Los mismos se clasifican de la siguiente manera:

- mochila_chica_nxx_no_conflict.txt: instancia KP01 donde la capacidad es reducida y entran varios ítems dentro de la mochila.
- mochila_apretada_nxx_no_conflict.txt: instancia KP01 donde la capacidad es significativa pero entran pocos ítems dentro de la mochila.
- costo_peso_correlaciona_nxx_cycle.txt: instancia KP01wCG donde algunos ítems presentan conflictos con otros de forma aleatoria.
- costo_peso_correlaciona_nxx_star.txt: instancia KP01wCG donde un único ítem presenta conflicto con algunos ítems de la mochila.

Experimento 1: ¿Hasta qué tamaño de instancias podemos abordar para el KP01wCG, y en qué tiempos de cómputo?

Variable independiente (VI): tamaño de instancias
Variable dependiente (VD): tiempo de ejecución
Variables controladas: algoritmo utilizado (BT o FB), condiciones de ejecución
Archivos: costo_peso_correlaciona_nxx_cycle.txt, costo_peso_correlaciona_nxx_star.txt
Procedimiento: ejecutar los algoritmos con un tamaño de instancia cada vez mayor, hasta que supere un timeout determinado (10 min).

Experimento 2: Los algoritmos implementados en C++ para KP01wCG se ejecutan en menor tiempo que los de Python, bajo las mismas condiciones de input.

Variable independiente (VI): lenguaje de programación (C++ o Python)
Variable dependiente (VD): tiempo de ejecución del algoritmo
Variables controladas: algoritmo utilizado (BT), mismo archivo de instancia, condiciones de ejecución
Archivos: costo_peso_correlaciona_nxx_cycle.txt, costo_peso_correlaciona_nxx_star.txt
Procedimiento: ejecutar el algoritmo de BackTracking con KP01wCG en C++ y Python, utilizando un mismo archivo de instancia.

Experimento 3: El algoritmo de BackTracking resuelve el problema KP01wCG en menor tiempo que el de FuerzaBruta, bajo las mismas condiciones de input.

Variable independiente (VI): algoritmo (Backtracking o Fuerza Bruta)
Variable dependiente (VD): tiempo de ejecución
Variables controladas: lenguaje (C++), mismo archivo de instancia, condiciones de ejecución
Archivos: costo_peso_correlaciona_nxx_cycle.txt, costo_peso_correlaciona_nxx_star.txt
Procedimiento: ejecutar los algoritmos de BackTracking y Fuerza Bruta para KP01wCG en C++, usando un mismo archivo de instancia.

Experimento 4: El algoritmo de DynamicProgramming resuelve el problema KP01 en menor tiempo que el de BackTracking, bajo las mismas condiciones de input.

Variable independiente (VI): algoritmo (DynamicProgramming o Backtracking)
Variable dependiente (VD): tiempo de ejecución
Variables controladas: lenguaje (C++ o Python), mismo archivo de instancia (KP01), condiciones de ejecución
Archivos: mochila_chica_nxx_no_conflict.txt, mochila_apretada_nxx_no_conflict.txt
Procedimiento: ejecutar los algoritmos de DynamicProgramming y BackTracking para KP01 en C++, utilizando un mismo archivo de instancia.

6.1. Metodología

Cada uno de los experimentos fue ejecutado en una laptop con las siguientes características:

- ➔ Hardware: Intel Core i5-1155G7, 16 GB RAM, SSD 474 GB
- ➔ Sistema Operativo: Windows 11
- ➔ Software: Python 3.13.2, C++ con Docker 4.40.0

Para compilar y ejecutar los experimentos, se debe incluir los siguientes SRC y HEADERS en el archivo Makefile:

SRC = CodigoExperimento.cpp BacktrackingKP01wCG.cpp KP01withCGInstance.cpp Solution.cpp Graph.cpp BruteForceKP01wCG.cpp DynamicProgrammingKP01.cpp
HEADERS = KP01withCGInstance.h Solution.h DynamicProgrammingKP01.h BacktrackingKP01wCG.h BruteForceKP01wCG.h Graph.h

7. Resultados y discusión

Resultado 1: ¿Hasta qué tamaño de instancias podemos abordar para el KP01wCG, y en qué tiempos de cómputo?

En base a los datos observados, realizamos una tabla para visualizar los tiempos de corte:

Instancia	FB C++ (s)	BT C++ (s)	BT Python (s)
costo_peso_correlaciona_n28_cycle	48.964	0.048	0.048
costo_peso_correlaciona_n30_cycle	817.614	0.032	0.068
costo_peso_correlaciona_n32_cycle	TIMEOUT	0.116	0.116
costo_peso_correlaciona_n36_cycle	TIMEOUT	0.437	0.437
costo_peso_correlaciona_n40_cycle	TIMEOUT	0.149	0.431
…			
costo_peso_correlaciona_n28_star	20.537	1.097	1.852
costo_peso_correlaciona_n30_star	86.304	11.041	15.420
costo_peso_correlaciona_n32_star	TIMEOUT	34.028	122.051
costo_peso_correlaciona_n36_star	TIMEOUT	105.289	150.452
costo_peso_correlaciona_n38_star	TIMEOUT	237.121	351.941
costo_peso_correlaciona_n40_star	TIMEOUT	TIMEOUT	TIMEOUT

Tabla 1. Tiempos de ejecución de los algoritmos de BruteForce y BackTracking para el problema de KP01wCG.

Como una primera observación, evidentemente el algoritmo de fuerza bruta es muchísimo más lento que el de backtracking, pues ya con n = 32, la ejecución supera al límite de tiempo determinado (10 min) en ambos tipos de instancia. Aún más, notamos también que el último archivo que pudo procesarse, tanto para cycle como para star, tiene un tiempo de ejecución bastante elevado. Esto demuestra la exponencialidad del algoritmo, y aunque podemos observar una cierta tendencia similar para el caso de backtracking en las instancias de star, lo cierto es que la ejecución pudo prolongarse hasta un tamaño mayor con n = 38.

No obstante, quizás lo más interesante de este experimento son los resultados obtenidos para el algoritmo de backtracking en las instancias de cycle. Incluso para un tamaño de n = 40, la ejecución apenas tarda poco más de 0.1 segundos. Aunque no tenemos un sustento relevante, creemos que esto se debe a la naturaleza de cycle. Como la asignación de conflictos es aleatoria, es más posible que haya mucho más incompatibilidades que en star, cuyos conflictos se concentran únicamente en un ítem. La verificación de esta hipótesis requeriría otras experimentaciones que puedan respaldarla.

Resultado 2: Los algoritmos implementados en C++ para KP01wCG se ejecutan en menor tiempo que los de Python, bajo las mismas condiciones de input.

Al igual que el experimento anterior, armamos una tabla con los datos obtenidos, midiendo el ratio de eficiencia que presenta la implementación de C++ frente a la de Python, utilizando BackTracking.

Instancia	BT Python (s)	BT C++ (s)	Relación de eficiencia (veces)
costo_peso_correlaciona_n20_cycle	0.003	0.005	**0.68**
costo_peso_correlaciona_n24_cycle	0.018	0.009	**2.06**
costo_peso_correlaciona_n28_cycle	0.048	0.025	**1.93**
costo_peso_correlaciona_n32_cycle	0.116	0.053	**2.20**
…			
costo_peso_correlaciona_n20_star	0.037	0.024	**1.55**
costo_peso_correlaciona_n24_star	1.517	0.903	**1.68**
costo_peso_correlaciona_n28_star	1.852	1.097	**1.69**

costo_peso_correlaciona_n32_star	49.351	34.028	**1.45**
costo_peso_correlaciona_n36_star	150.452	105.289	**1.43**

Tabla 2. Relación de eficiencia (veces) entre las implementaciones de C++ y Python para el algoritmo de BackTracking con KP01wCG.

Dados los datos y el ratio de comparación, en principio validamos la hipótesis planteada. Como se observa en la *Tabla 2*, en instancias de cycle C++ supera a Python con ratios que van desde 1.42 hasta 2.20 veces. En instancias tipo star, si bien las diferencias son más suaves, aún se mantiene la superioridad de C++, con ratios que oscilan entre 1.43 y 1.69.

Para ampliar la observación, realizamos unos gráficos que comparan las ejecuciones de BackTracking usando C++ y Python, para instancias de cycle y star.

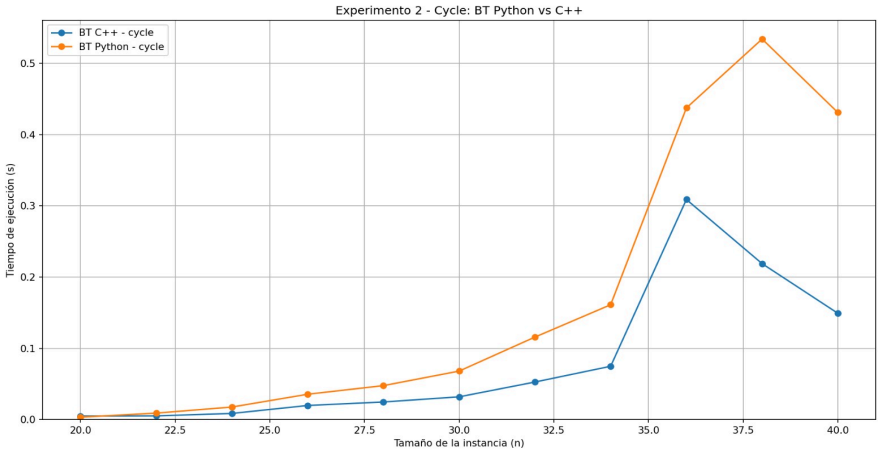

Fig. 3 Gráfico que compara las ejecuciones de BT usando C++ y Python, para las instancias tipo cycle

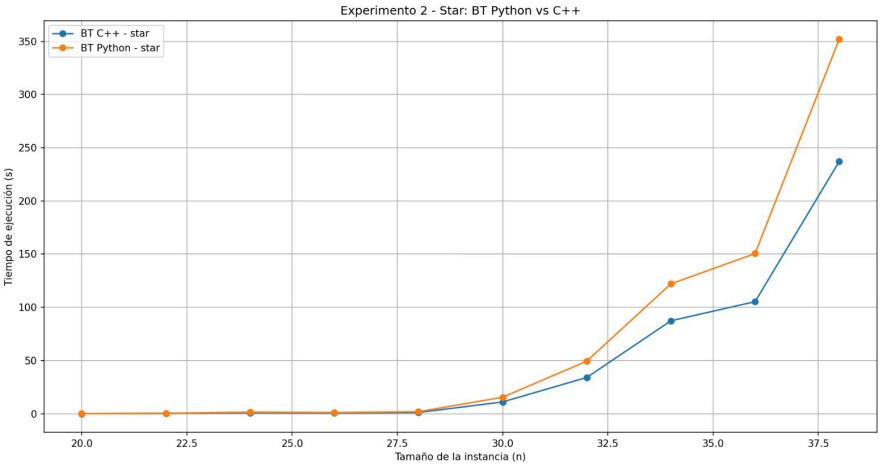

Fig. 4 Gráfico que compara las ejecuciones de BT usando C++ y Python, para las instancias tipo star

De la misma manera, los gráficos parecen reforzar lo visto en la *Tabla 2*. Notoriamente, las curvas muestran un crecimiento progresivo, donde la de Python se mantiene constantemente por encima de la de C++, con una separación creciente a medida que aumenta el tamaño de la instancia.

Curiosamente, si prestamos atención al gráfico de cycle, notaremos que los tiempos de ejecución disminuyen para las últimas instancias. Estos son casos de outliers, y hay varios factores que pueden explicarlo. Quizás se deba a que las podas fueron extremadamente eficientes, quizás se deba a la alta cantidad de conflictos entre ítems, pero en cualquier caso, se sigue manteniendo nuestra hipótesis, dado que Python continúa ejecutando más lento que C++.

Además, a partir de los gráficos, podemos visualizar de mejor manera la exponencialidad del BT. Las curvas tienden a comportarse como una función exponencial, aunque el fenómeno se vea más marcado para las instancias de star.

Resultado 3: El algoritmo de BackTracking resuelve el problema KP01wCG en menor tiempo que el de FuerzaBruta, bajo las mismas condiciones de input.

Instancia	BFKP01wCG (s)	BTKP01wCG (s)	Relación de eficiencia (veces)
costo_peso_correlaciona_n20_cycle	0.435	0.005	**89.16**
costo_peso_correlaciona_n22_cycle	0.738	0.005	**145.40**
costo_peso_correlaciona_n24_cycle	2.441	0.009	**286.63**
costo_peso_correlaciona_n26_cycle	11.465	0.020	**582.30**
costo_peso_correlaciona_n28_cycle	48.964	0.025	**1950.07**
…			
costo_peso_correlaciona_n20_star	0.092	0.024	**3.82**
costo_peso_correlaciona_n22_star	0.352	0.297	**1.18**
costo_peso_correlaciona_n24_star	1.475	0.903	**1.63**
costo_peso_correlaciona_n26_star	4.990	0.695	**7.18**
costo_peso_correlaciona_n28_star	20.537	1.097	**18.72**

Tabla 3. Relación de eficiencia (veces) entre los algoritmos de BackTracking y BruteForce para el problema de KP01wCG.

Observando la tabla, concluimos en que en la comparación entre BackTracking y BruteForce, ambos en C++, vuelve a mostrar una superioridad contundente de BT, llegando a ratios de eficiencia de 1950.07 (!). Retomando el análisis del primer experimento, la eficiencia es tanta que para tamaños de instancia mayores a 30, ni siquiera podemos calcularla tomando un timeout de 10 minutos. De este modo, los resultados respaldan positivamente nuestra hipótesis, y demuestra la gran mejora y capacidad de BackTracking para descartar ramas del árbol gracias a las podas.

Resultado 4: El algoritmo de DynamicProgramming resuelve el problema KP01 en menor tiempo que el de BackTracking, bajo las mismas condiciones de input.

Instancia	BTKP01 C++ (s)	DPKP01 C++ (s)	Relación de eficiencia (veces)
mochila_chica_n10_no_conflict	0.003	0.004	**0.69**
mochila_chica_n20_no_conflict	0.560	0.013	**42.42**
mochila_chica_n30_no_conflict	175.022	0.050	**3471.54**
mochila_chica_n40_no_conflict	TIMEOUT	0.121	**∞**
…			

mochila_apretada_n20_no_conflict	0.005	0.153	**0.03**
mochila_apretada_n40_no_conflict	0.005	0.273	**0.02**
mochila_apretada_n60_no_conflict	0.017	0.416	**0.04**
mochila_apretada_n80_no_conflict	0.015	0.575	**0.03**

Tabla 4. Relación de eficiencia (veces) entre los algoritmos de DynamicProgramming y BackTracking para el problema de KP01, utilizando C++.

Para nuestra última hipótesis, empezando por el análisis en C++, desde la *Tabla 4* observamos inicialmente una validación positiva de la misma en las instancias de mochila chica. Con DP, el tiempo de ejecución resulta muchísimo menor con respecto a BT. Incluso para casos de n > 30, el algoritmo de BackTracking ni llega a procesarse dentro del timeout fijado de 10 minutos. Mientras tanto, con la programación dinámica la ejecución no tarda mucho más que 0.1 segundos (¡Qué locura!).

Sin embargo, las instancias de mochila apretada revelan un comportamiento totalmente contrario. El algoritmo de DynamicProgramming se ejecuta con notoria lentitud frente a BackTracking. ¿Por qué? Esto se debe a la anatomía de este tipo de instancias, cuya capacidad alcanza a C = 1000000. Como el algoritmo de DP consta de rellenar una matriz en O(N.C + N.log(N)), considerando un valor de C de tal magnitud, la ejecución empieza a escalar. Y aunque ciertamente el tiempo no llega todavía al segundo, es innegable que BackTracking posee la ventaja. ¿Qué significa esto? ¿Rechazamos nuestra hipótesis? La realidad es que la validación de la misma depende enormemente del tipo de instancias que se desea ejecutar.

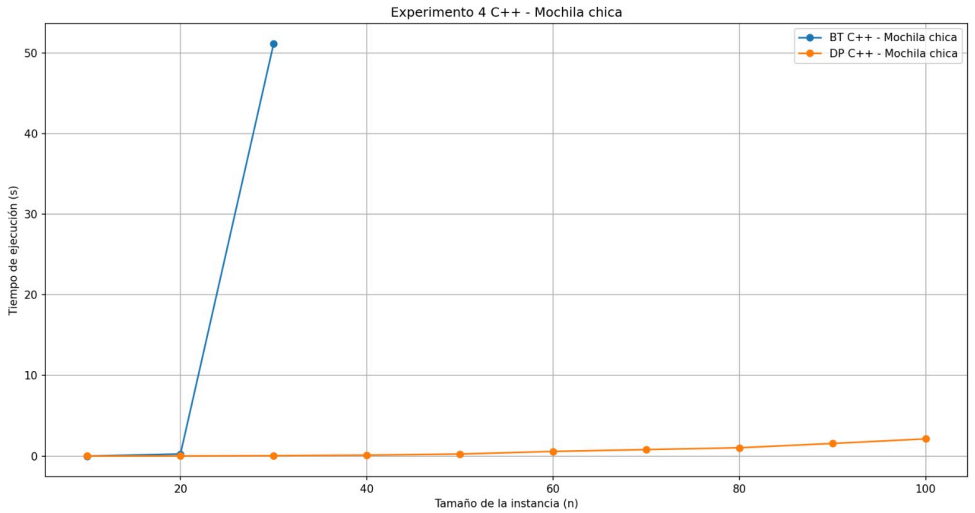

Fig. 5 Gráfico que compara las ejecuciones de BT y DP usando C++, para las instancias tipo mochila chica

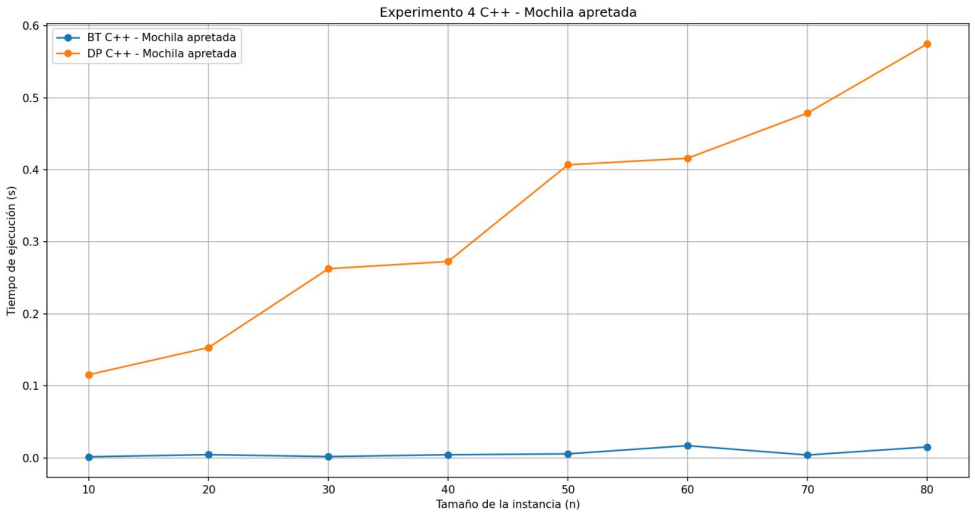

Fig. 6 Gráfico que compara las ejecuciones de BT y DP usando C++, para las instancias tipo mochila apretada

A través de los gráficos, podemos ver cómo se reafirma lo propuesto anteriormente. Para la mochila chica, DP lleva la ventaja tomando valores de procesamiento mucho más chicos, mientras que para la mochila apretada, BT tiene un tiempo de ejecución notoriamente menor.

Asimismo, algo interesante de estos gráficos es que nos permite visualizar el crecimiento pseudo polinomial de Programación Dinámica, pronunciándose de mejor modo para la mochila apretada.

Instancia	BTKP01 Python (s)	DPKP01 Python (s)	Relación de eficiencia (veces)
mochila_chica_n10_no_conflict	0.0010	0.0282	**0.04**
mochila_chica_n20_no_conflict	0.5997	0.4760	**1.26**
mochila_chica_n30_no_conflict	60.3159	1.3366	**45.12**
mochila_chica_n40_no_conflict	TIMEOUT	3.1378	∞
mochila_chica_n70_no_conflict	TIMEOUT	19.418	∞
mochila_chica_n100_no_conflict	TIMEOUT	69.282	∞
…			
mochila_apretada_n20_no_conflict	0.0001	2.3971	**0.000045**
mochila_apretada_n40_no_conflict	0.0004	4.8836	**0.0000723**
mochila_apretada_n60_no_conflict	0.0015	7.3155	**0.000199**
mochila_apretada_n80_no_conflict	0.0036	9.9609	**0.000365**

Tabla 5. Relación de eficiencia (veces) entre los algoritmos de DynamicProgramming y BackTracking para el problema de KP01, utilizando Python.

Luego, para la implementación en Python, el comportamiento es similar. Para la mochila chica, los tiempos de ejecución de BackTracking se disparan a partir de n = 30, mientras que Programación Dinámica se resuelve en tiempos relativamente chicos. Por otro lado, para la mochila apretada, la observación es justamente la contraria: Programación Dinámica se procesa en tiempos cada vez más elevados.

Otra observación importante es la comparación que podemos realizar entre las ejecuciones de DP utilizando C++ y Python. Revisando el gráfico de la Figura 5, aún para tamaños de instancias grandes como n=100, los tiempos de ejecución en C++ con DP no superan los cinco segundos. No obstante, la implementación en Python llega a alargarse incluso hasta los 69.282 segundos, notoriamente mayor a la de C++. Esto resulta un respaldo crítico para el primer experimento, e incluso nos confirma que, aún con una complejidad teórica mayor, C++ se ejecuta en menor tiempo.

8. Conclusión

A partir de los experimentos realizados y los resultados obtenidos, se confirma la validez de las hipótesis planteadas, con una cierta sutileza en la última.

En primer lugar, se evidencia que las implementaciones en C++ resultan consistentemente más eficientes que sus equivalentes en Python, especialmente en instancias grandes o con estructuras de conflicto complejas. Esto se debe al control más detallado de recursos y la ejecución compilada de C++, lo cual permite gestionar con mayor rapidez estructuras de memoria extensas, como las requeridas en KP01wCG.

Por otro lado, se observó que el algoritmo de Backtracking supera ampliamente al de Fuerza Bruta, no solo en la teoría sino también en la práctica. La incorporación de podas efectivas permitió reducir significativamente el espacio de búsqueda, especialmente en instancias con mayores restricciones de conflictos, como las instancias tipo cycle.

Finalmente, la comparación entre Backtracking y Programación Dinámica para KP01 demostró que dependiendo del tipo de instancia, el algoritmo puede tener o no ventaja sobre el otro. Mientras que la eficiencia de DP depende del tamaño de capacidad, BT logra mantenerse en tiempos aceptables si las podas resultan efectivas.

En conjunto, estos hallazgos no solo validan las hipótesis iniciales, sino que también permiten reflexionar sobre la importancia de elegir adecuadamente el enfoque algorítmico y el lenguaje de programación según las características del problema a resolver.

Referencias bibliográficas

[1] K. Jansen (1998). An approximation scheme for bin packing with conflicts. *Lecture Notes in Computer Science*, 1432:35–46.
[2] Ramesh Kumar C (2019). A Comparison between Python and C++. Paper recuperado de https://www.jetir.org/papers/JETIREQ06006.pdf