BRANCH & BOUND

ANÁLISIS Y DISEÑO DE ALGORITMOS.

Práctica Final

Memoria

Elvi Mihai Sabau - 51254875L

ÍNDICE

1. Estructuras de datos.
   1.1. Nodo.
   1.2. Lista de nodos vivos.

2. Mecanismos de poda.
   2.1. Poda de tuplas no factibles.
   2.2. Poda de tuplas no prometedoras.

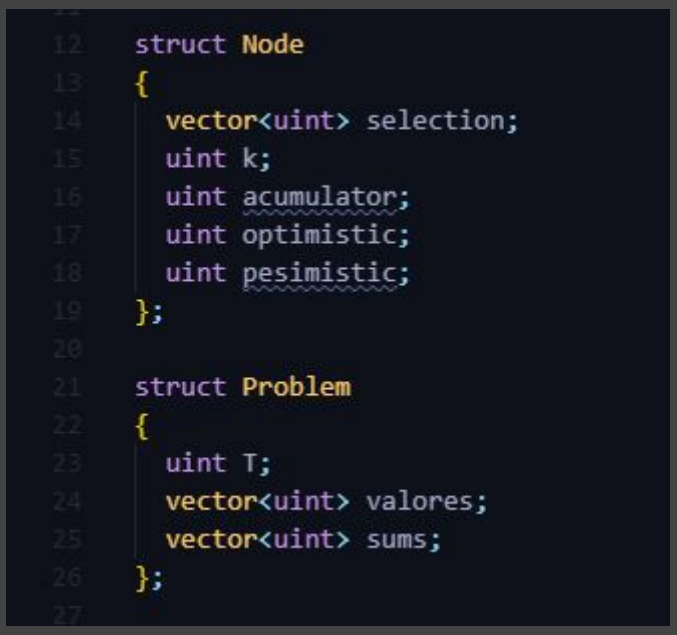
3. Cotas pesimistas y optimistas.
   3.1. Cota pesimista inicial (Inicialización).
   3.2. Cota pesimista del resto de nodos.
   3.3. Cota optimista.

4. Otros medios empleados para acelerar la búsqueda.

5. Estudio comparativo de distintas estrategias de búsqueda.

6. Tiempos de ejecución.

Estructuras de datos.



```
struct Node
{
  vector<uint> selection;
  uint k;
  uint acumulator;
  uint optimistic;
  uint pesimistic;
};

struct Problem
{
  uint T;
  vector<uint> valores;
  vector<uint> sums;
};
```

Usaré structs para manejar de manera más cómoda y organizada los datos en el problema.

Nodo.

```
struct Node
{
  vector<uint> selection;
  uint k;
  uint acumulator;
  uint optimistic;
  uint pesimistic;
};
```

Para manejar los datos como nodo, he usado un struct, siguiendo el ejemplo que hay en las transparencias actualizadas, en el nodo me guardo la selección actual de los valores, la siguiente decisión, la acumulación para poder comparar entre nodos, y su valor en cota optimista y pesimista.

Lista de nodos vivos.

```
Node initial_node()
{
  Node inicial;
  inicial.k = 0;
  inicial.acumulator = 0;
  return inicial;
}
```

```
priority_queue<Node> q;
Node inicial = initial_node();
q.push(inicial);
```

Para la lista de nodos vivos, he decidido usar una priority queue, donde me voy guardando los nodos que podrían ser expandidos.

```
// Sobrecarga para la decisión interna del priority queue
bool operator<(const Node &node1, const Node &node2)
{
  return node1.k > node2.k;
}
```

A su vez, me he hecho una sobrecarga del operador < entre nodos para poder sacar el máximo partido a la función *top* del priority queue.

Mecanismos de poda.

Poda de tuplas no factibles.

```
bool is_feasible(const Problem &problem, const Node &node)
{
  return node.acumulator <= problem.T;
}
```

La manera en la que podo si una tupla no es factible es similar a la que se ha usado hasta las prácticas actuales, comparamos si el valor total de dicho nodo es menor o igual al límite del precio, en caso de serlo, es factible, en caso de no serlo, no lo es.

```
if (is_feasible(problem, nodeExpanded))
{
  uint pessimisticSol = pessimistic_solution(problem, node);
  node.pesimistic = pessimisticSol;

  if (is_better(pessimisticSol, current_best))
  {
    current_best = pessimisticSol;
    best_sol_updated_from_pessimistic_unfinished++;
  }

  if (is_promissing(problem, nodeExpanded, current_best))
  {
    q.push(nodeExpanded);
    added++;
  }
  else
    was_promissing_discarted = promissing_discarted++;
}
else
  feasible_discarted++;
```

Los nodos no factibles los descarto directamente, ya que no es necesario contemplar las situaciones en las cuales el valor total de los objetos a escoger de la tienda supera la cantidad límite.

Poda de tuplas no prometedoras.

```
bool is_promissing(const Problem &problem, Node &node, const Solution &current_best)
{
  node.optimistic = node.acumulator + add_rest(problem, node);
  return node.optimistic > current_best.acumulator;
}
```

Para la poda de tuplas no prometedoras, lo que hago para decidir si es prometedora o no es comprobar si el nodo actual tiene una suma mayor que el mejor caso hasta el momento. Esto siempre teniendo en cuenta que dicho nodo es factible.

```
if (is_promissing(problem, nodeExpanded, current_best))
{
  q.push(nodeExpanded);
  added++;
}
// Mando tutoria para preguntar si es necesaria dicha comprobación.
// was_promissing_discarted;
else
  promissing_discarted++;
```

Y para podar lo único que hago es, en caso no ser prometedor, no añadirlo a mi cola de nodos vivos.

Cotas pesimistas y optimistas.

```
uint pessimistic_solution(const Problem &problem, const Node &node)
{
  uint sol;
  sol = node.acumulator;

  for (uint i = node.k; i < problem.valores.size(); i++)
  {
    if (problem.valores[i] + sol <= problem.T)
      sol += problem.valores[i];
  }
  return sol;
}
```

```
bool is_promissing(const Problem &problem, Node &node, const uint &current_best)
{
  node.optimistic = node.acumulator + add_rest(problem, node);
  return node.optimistic > current_best;
}
```

Cota pesimista inicial (Inicialización).

```
Node initial_node()
{
  Node inicial;
  inicial.k = 0;
  inicial.acumulator = 0;
  return inicial;
}
```

```
uint branch_and_bound(const Problem &problem)
{
  priority_queue<Node> q;
  Node inicial = initial_node();
  q.push(inicial);

  uint current_best = pessimistic_solution(problem, inicial);
```

La cota pesimista inicialmente la calculo haciendo una solución voraz de todo el problema.

Cota pesimista del resto de nodos.

```
while (!q.empty())
{
  Node node = q.top();
  q.pop();

  if (is_leaf(problem, node))
  {
    finished++;
    uint currentSol = solution(node);

    if (is_better(currentSol, current_best))
    {
      current_best = currentSol;
      best_sol_updated_from_finished++;
    }

    continue;
  }

  vector<Node> expandedBranches;
  expandedBranches = expand(problem, node);

  for (Node nodeExpanded : expandedBranches)
  {
    expanded++;
    if (is_feasible(problem, nodeExpanded))
    {
      uint pessimisticSol = pessimistic_solution(problem, node);
      node.pesimistic = pessimisticSol;

      if (is_better(pessimisticSol, current_best))
      {
        current_best = pessimisticSol;
        best_sol_updated_from_pessimistic_unfinished++;
      }
```

```
bool is_better(const uint &solution1, const uint &solution2)
{
  return solution1 > solution2;
}
```

Para el resto de nodos, aplicó el mismo método voraz, pero cada vez para el nodo mejor que sacamos de la cola, y comprobamos si dicha solución es mejor que la actual, de ser así, actualizamos la solución actual.

Cota optimista.

```
bool is_promissing(const Problem &problem, Node &node, const uint &current_best)
{
  node.optimistic = node.acumulator + add_rest(problem, node);
  return node.optimistic > current_best;
}
```

La cota optimista la calculo implícitamente cuando comprobamos si es prometedora, revisando si la suma actual, más la siguiente voraz, es más ajustada que la suma actual.

```
if (is_feasible(problem, nodeExpanded))
{
  uint pessimisticSol = pessimistic_solution(problem, node);
  node.pesimistic = pessimisticSol;

  if (is_better(pessimisticSol, current_best))
  {
    current_best = pessimisticSol;
    best_sol_updated_from_pessimistic_unfinished++;
  }

  if (is_promissing(problem, nodeExpanded, current_best))
  {
    q.push(nodeExpanded);
    added++;
  }
  else
    was_promissing_discarted = promissing_discarted++;
}
else
  feasible_discarted++;
```

De este modo nos ahorramos calcular los nodos que no son factibles directamente.

Otros medios empleados para acelerar la búsqueda.

```
sort(problem.valores.begin(), problem.valores.end());
reverse(problem.valores.begin(), problem.valores.end());
// random_shuffle(problem.valores.begin(), problem.valores.end());
```

Se ha probado reordenando el vector de precios (valores) de otras maneras que no sea de mayor a menor, (de menor a mayor, split, y porqué no de manera aleatoria), pero el resultado concluyente fué que para ciertas pruebas, dicha estrategia funcionaba mejor y peor para todas las pruebas, alterando el tiempo medio de todas, y por ende, haciendo el algoritmo bastante inestable con respecto al tipo de problema que se recibe. Mi objetivo es que el algoritmo se pueda comportar de una manera más o menos homogénea para todas las pruebas, así que se descarto esta estrategia.

```
for (Node nodeExpanded : expandedBranches)
{
  expanded++;
  {
    uint pessimisticSol = pessimistic_solution(problem, node);
    node.pesimistic = pessimisticSol;

    if (is_better(pessimisticSol, current_best))
    {
      current_best = pessimisticSol;
      best_sol_updated_from_pessimistic_unfinished++;
    }

    if (is_promissing(problem, nodeExpanded, current_best))
    {
      q.push(nodeExpanded);
      added++;
    }
    else
      was_promissing_discarted = promissing_discarted++;
  }
}
```

También se ha probado ignorando la restricción que no permita a un nodo continuar ser comprobado si es prometedor, aunque no sea factible, es menos lento porque no tiene que hacer dicha comprobación para cada nodo, pero hay más podas.

Estudio comparativo de distintas estrategias de búsqueda.

[NO IMPLEMENTADO]

Tiempos de ejecución.

Fichero:	Tiempo (ms):
1_bb	0,096
10_bb	0,018
20_bb	0,925
30_bb	183,227
40_bb	3650,55
45_bb	1310,40
50_bb	0,693
100_bb	0,116
200_bb	0,033
1k_bb	0,628
2k_bb	0,028
3k_bb	0,137

- Nótese el pico en las pruebas 30, 40, 45, esto se debe a que el valor es muy alto, y la cota tiene poco efecto. En estos caso, haber levantado la restricción, habría ayudado en dichos problemas.