1 Introdução

O desafio do Laboratório é resolver o problema do empacotamento com divisórias por meio de algoritmos de *backtracking* e *branch and bound*, analisando quantitativamente seus desempenhos.

2 Backtracking

backtrack_rec é a função que faz recursão dentre as possibilidades de alocação de usuários na banda. Para isso, primeiro simulamos o que acontece se tentarmos colocar mais um usuário na banda. Então, usamos a função validator para validar a possibilidade. Se a possibilidade vale, testamos um backtracking para ela. Senão, apenas tentamos um backtracking sem o usuario, dada a solução candidata melhor até o momento.

3 Branch and Bound

A técnica branch and bound ordena todos os pacotes enfileirados e monta uma árvore com ramificação binária composta pela escolha (ou não) de um certo item. Um critério de poda preciso e uma mescla de busca em largura e profundidade foram obtidos em busca de uma rápida aproximação do valor ótimo.

A ordenação é feita pelo valor relativo, do maior para o menor. O critério de poda usa dos seguintes fatos: Seja K o conjunto de classes presentes na solução atual. Seja r_K o número de pacotes cuja classe pertence ao conjunto K mas não estão na solução atual. A melhor situação possível é que todos os r_K primeiros pacotes (ordenados por valor relativo) ainda não presentes na solução tenham classe no conjunto K, pois isso evita a necessidade de uma divisória nova. Após usar todos esses pacotes, o melhor que pode acontecer é que os pacotes restantes, em ordem de valor relativo, pertençam respectivamente às classes c_1, c_2, ..., c_k, sendo o número de elementos da classe c_1 maior que o da c_2 e assim por diante. Esse cenário garante o menor uso de divisórias possível. Assim, o critério de pode assume esse cenário ideal e calcula o valor máximo que pode ser obtido pela ramificação. Se este ainda for menor que o atual máximo, não faz sentido continuar a busca naquele ramo e então é feita a poda.

Iterative deepening depth-first search (IDDFS) é feita para a parte mais ao topo da árvore com o objetivo de já obter uma boa aproximação do valor ótimo. Assim, começa-se da profundidade 1 e vai-se até um certo limite (definido aqui como duas vezes o número de classes). Essa heurística deu bons resultados na prática, mostrando um bom compromisso entre tempo e aproximação do valor ótimo.

`bnb` implementa o algoritmo branch and bound. A rotina ordena os elementos por valor relativo e então chama `_bnb`, rotina recursiva que faz a busca de fato dos valores.

4 Complexidade

O grafo foi implementado como uma lista de adjacências. As regiões a serem vigiadas foram implementadas como uma matriz binária, com 1 se e só se o ponto (i, j) é uma região alfa. Seja V o número de vértices de um grafo e E o número de arestas do mesmo e l o número de linhas e c o número de colunas da matriz das regiões.

A linha 2 é simbólica pois na prática pode-se determinar qual o lado da bipartição de um nó só com valores booleanos. O *loop* da linha 4 é basicamente uma iteração sobre cada ponto da matriz, checando ao seu redor para adjacências. O mapeamento de um ponto para um nó da linha 5 é feito em tempo constante com uma matriz de mapeamento. As linhas 8 e 9 checam se as arestas já estão no grafo de capacidades antes de adicioná-las. Isso é feito em tempo constante com um vetor. Assim, a transformação leva tempo $O(lc)$. A rede retornada é a entrada do algoritmo `EDMONDS-KARP` que foi implementado como esperado com `BFS` e então leva tempo $O(VE^2)$ com V, E das capacidades. Para se obter a resposta do problema original a partir do fluxo máximo, simplesmente calcula-se: $n = |C| - 2 - |f|$, onde f, C são o fluxo e capacidade, respectivamente. O grafo de capacidades é construído de tal forma que ele tenha $lc + 2$ nós: pontos que não são adjacências simplesmente não se conectam a nada. Isso leva tempo constante. Assim, o tempo final do algoritmo é $O(lc + VE^2) = O(l^3c^3)$ pois $V, E = O(lc)$.