Relatório do Projeto 1 – Bactracking

Disciplina: Projeto de Algoritmos - SCC5900

Aluno: Cézanne Alves – 10846548

Este relatório visa descrever detalhes da implementação do aluno do algoritmo de backtracking para a solução do problema do Futoshiki e comparar o desempenho do algoritmo com e sem as heurísticas de checagem adiante e MRV. O algoritmo abaixo será usado pra discutir a implementação

O molde de um algoritmo de backtracking é quase sempre o mesmo, a implementação varia quase sempre em como se faz os passos A, B e C.

O passo A é relacionado ao MRV, e o C só existe quando há checagem adiante. O passo B exite em toda implementação de backtracking.

O problema do futoshiki é combinatório, e seu espaço de busca é de $O(c^n)$, onde c é a quantidade de valores (algarismos) disponíveis e $n = d^2$ é a quantidade de células. Observa-se que $\boldsymbol{c = d = \sqrt{n}}$, mas serão usados alternadamente para clareza dependendo do contexto.

O passo B, inerente ao backtracking, pode ser feito iterando por cada valor e checando sua validade, ou mantendo alguma estrutura que permite recuperar o domínio válido de cada variável e iterar sobre esses valores.

A iteração de B tem custo $O(c)$, mas fica amortizado para $O(1)$ pois cada recursão corresponde a apenas uma iteração dessas no nó pai. Outra forma de ver isso é: $c \times c^n = c^{n+1} \in O(c^n)$. Quase qualquer operação que se faça na recursão também será dominada pelo tempo de busca na arvore, e tudo que podemos fazer é manter a complexidade de cada recursão o menor possível, para manter o percurso pela arvore rápido. Sendo assim, analisaremos aqui a complexidade das operações internas da recursão, com exceção da iteração sobre c, inerente ao backtracking, que será considerado amortizado $O(1)$

Com isso, até então, nossa recursão tem custo $O(1)$, resta identificar uma forma de validar os valores, ou manter uma estrutura de valores válidos para calcular o custo total de B na recursão. Antes de apresentar a solução implementada, discutiremos algumas soluções ingênuas para os passos A, B e C.

A forma mais ingênua de realizar o passo B seria, para cada valor c do domínio, verificar se infringe uma das $r \in O(n)$ regras, e adicionalmente, percorrer sua linha e coluna para verificar se c é único nelas, somando $r + 2d \in O(n)$. Uma solução melhor, seria armazenar em 4 tabelas binárias a informação de se a célula deve ser menor que cada um dos 4 vizinhos, computando em tempo constante, e depois percorrendo a coluna e a linha, totalizando $2d \in O(\sqrt{n})$.

```
backtrack():
  se preencheu: retorna verdadeiro
  busca próxima variável vazia v //A
  para cada valor c válido em c:  //B
    v:=c
    se (checagem_adiante = ok): //C
      backtrack()
  v:=vazio
  retorna falso
```

Seria possível também manter o domínio pré-computado de cada variável, mas sempre que atualizássemos uma variável seria necessário atualizar o domínio de todas as variáveis da mesma linha e coluna ($O(d)$), sendo necessário usar um contador para cada valor, pois se um vizinho deixar de restringir um valor, pode ser que outro ainda o restrinja, e uma restrição de vizinhança pode tornar necessário atualizar até c-1 contadores ($O(c)$), tornando o custo da manutenção desses domínios $4 \times c + 2d \in O(c + d)$.

No passo C, caso a solução adotada no passo B tenha sido pré-computar o domínio, basta aproveitar essa estrutura e checar se alguma das O(n) variáveis vazias está sem domínio, ou melhor, checar apenas as variáveis de mesma linha e coluna da que acabou de ser alterada totalizando $O(d \times c)$ (no caso de apenas o último ser válido).

Se a solução adotada em B for validar os valores durante a iteração e a validação de um valor tiver custo O(d) (por checar toda a linha e a coluna) a verificação adiante pode custar $O(d^2 \times c)$, mesmo checando apenas as variáveis da linha e coluna da variável alterada.

O passo A exige que seja feito (em algum momento) o cômputo da cardinalidade do domínio de toda variável, e a busca do maior valor. Aqui percebe-se que não é ideal fazer isso sem nenhuma estrutura que permita acelerar o processo, pois custaria, para cada variável(O(n)) percorrer sua coluna e linha(O(d)), seus vizinhos e restrições de desigualdade(O(c)), e contar a quantidade de valores restantes(O(c)), totalizando $ndc + c \in O(n\sqrt{nd})$.

A solução adotada faz uso pesado de bitsets e de sua complexidade de operações aproximadamente constante. As operações de definir um bit, uma faixa de bits, união, disjunção, e remover o menor bit 1 são implementáveis em tempo constante com operadores binários num tipo inteiro. E, num compilador e hardware com suporte, a operação de retorno do índice do menor bit 1, e a contagem de bits um, ocorrem em apenas uma instrução do processador, **tzcnt** (*trailing zeros count*) e **popcnt** (*pop count*) respectivamente nos Intel e AMD x64 recentes. Essas operações na verdade são dependentes do tamanho do bitset, assim como a operação de soma é, caso a quantidade de bits seja maior que a mantissa. Mas como essas operações são processadas em uma instrução para um c de até 64, tamanho de entrada muito além do tratável para c^{2c}, serão consideradas aqui como constantes.

Para o cômputo do domínio de uma variável, que é feito sob demanda, um grafo é mantido onde cada célula é ligada aquelas com as quais tenham alguma restrição de desigualdade, as quais são computadas em 4 operações de bitset. Além disso cada linha e coluna mantém um bitset dos seus valores disponíveis e atualiza-se o domínio da variável com 2 ***and***’s lógicos. Assim, tanto o cômputo do domínio de uma variável, quanto a atualização da estrutura necessária (bitsets das linhas e colunas) é feito em $\boldsymbol{O(1)}$. Convenientemente, a iteração B no domínio não testa valores inválidos.

Com isso, a complexidade amortizada do laço B continua constante, assim como a complexidade de toda uma recursão do **backtracking sem heurísticas** em **O(1)**.

Aproveitando a estrutura acima, a Checagem Adiante é feita computando a cardinalidade dos bitsets (O(1)) para cada célula da linha e da coluna daquela que foi alterada. Totalizando o custo amortizado de uma recursão do **backtracking com Checagem Adiante** em $\boldsymbol{O(d)}$.

Para o MRV, seria inviável ordenar os elementos, pois para cada alteração, segue uma consulta pelo mínimo. Também desconheço uma implementação de *heap* que permita alterar a chave de seus elementos tanto pra maior, quanto pra menor. Ainda, a manutenção numa ABB custaria uma remoção e inserção de até 2d variáveis quando o valor de alguma no tabuleiro fosse alterado, e $4\sqrt{n}\lg_2 n$ só é superado por n em torno de $n =$ 1897. Mesmo considerando que o percurso num array e na árvore teriam o mesmo custo constante, a busca numa lista estática com as variáveis vazias parece preferível para o tamanho limite da entrada. Assim, o custo amortizado de uma recursão do **backtracking com MRV** fica $\boldsymbol{O(n)}$.

As três versões foram testadas nos três últimos arquivos de teste disponibilizados até encontrar todas as soluções. Os gráficos mostram a soma da quantidade de atribuições e de tempo em segundos para todos os casos de cada arquivo.

Os casos do mrv_only são muito simples para apresentar divergência com as heurísticas: o sem, heurísticas, com FWC e com MRV, resolveram todos os casos com em média 30, 24 e 22. Nos outros casos é possível ver como as heurísticas diminuem a quantidade de iterações necessárias. Entretanto ao medir pelo tempo, o backtracking sem heurística com recursão $O(1)$ desempenha sempre melhor devido ao overhead das heurísticas. Percebemos também que usar apenas Checagem Adiante não é muito útil, pois demora mais iterações e mais tempo que o MRV (que implicitamente checa por variáveis com domínio vazio).

O arquivo de teste futoshiki-all tem casos muito difíceis, que não foram resolvidos nem com 10^9 iterações do backtracking puro. Para avaliar as técnicas nesse caso, cada caso de teste foi interrompido com 5 segundos.

Ademais, o mrv executa em média 300 mil recursões por segundo, A chegem adiante, 3 milhões, e o backtracking puro, 16 milhões