Trabalho Prático 1

Iago Zagnoli Albergaria - 2022069476

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG - Brasil

iagozag@gmail.com

1 Introdução

O problema proposto consiste em duas partes, uma sendo o complemento da outra. Na primeira parte do trabalho foi necessário implementar um avaliador de expressões lógicas, sendo a entrada do programa uma string de caracteres que representa a expressão a ser avaliada e, em seguida, os valores(0 ou 1) das variáveis. Já na segunda parte, o objetivo era implementar um programa que tinha possíveis quantificadores($\forall$ ou $\exists$) nos valores da expressão e avalia se a expressão lógica é satisfazível, ou seja, pode retornar verdadeiro para aquela expressão e valores / quantificadores e qual a solução para a entrada.

2 Método

O programa foi desenvolvido na linguagem C++ e compilado pelo compilador G++ da GNU Compiler Collection. O computador utilizado tem as seguintes especificações:

- Sistema Operacional: Arch Linux x86_64
- Kernel: 6.5.5-zen1-1-zen
- Processador: AMD Ryzen 7 5700U with Radeon Graphics (16) @ 4.372GHz
- RAM: 8,00GB

2.1. Estruturas de dados

A implementação do programa teve como base as estruturas de dados, pilha e árvore binária.

A pilha foi implementada com arranjo visando resolver a primeira parte do problema. Essa estrutura de dados foi escolhida pela forma como os elementos são inseridos e retirados da pilha(FIFO - First in First out) e serviu tanto para transformar a expressão lógica inicial para uma posfixa(forma de organizar uma expressão que leva em conta a ordem de precedência dos operadores), como também para resolver a expressão posfixa e retornar um valor booleano como resposta.

Com o objetivo de resolver o problema da satisfabilidade, foi necessário implementar uma árvore binária que tem vértices representando todos os possíveis valores para as variáveis, sendo as folhas todas as combinações possíveis trocando os quantificadores por 0 e 1. A razão da escolha de uma árvore binária e não outro tipo de árvore se dá por um motivo. Cada vértice que ainda tenha um quantificador na string sempre é dividido em dois filhos, sendo o filho da esquerda a mesma string com o '0' no lugar da variável quantificada e o filho

da direita com o '1' no lugar da variável quantificada. Ou seja, as folhas da árvore representam todas as 2^n possibilidades de substituição de n quantificadores na expressão lógica, trocando as variáveis por 0 ou por 1.

2.2. Classes

Foram utilizadas três classes no projeto, sendo elas a pilha, o vértice da árvore e a árvore binária em si.

A classe Stack consiste em uma simples implementação por arranjo de uma pilha com tamanho máximo de 10^6.

Cada função / variável está descrita por um comentário na imagem.

```
class Stack {
    private:
        int length;                         /**< Comprimento atual da pilha */
        static const int MAXLEN = 1e6+10;   /**< Tamanho máximo da pilha */
        T itens[MAXLEN];                    /**< Array para armazenar os itens da pilha */
    public:
        /**
        * @brief Construtor padrão da pilha.
        */
        Stack();

        /**
        * @brief Verifica se a pilha está vazia.
        * @return True se a pilha estiver vazia, False caso contrário.
        */
        bool empty();

        /**
        * @brief Insere um item no topo da pilha.
        * @param item - O item a ser inserido na pilha.
        */
        void push(T item);

        /**
        * @brief Retorna o item no topo da pilha.
        * @return O item no topo da pilha.
        */
        T top();

        /**
        * @brief Remove e retorna o item no topo da pilha.
        * @return O item removido do topo da pilha.
        */
        T pop();

        /**
        * @brief Limpa a pilha, redefinindo seu comprimento para 0.
        */
        void clear();
};
```

As classes NodeType e BinaryTree consistem na implementação da árvore binária, sendo a classe NodeType a definição de um vértice da árvore.

Cada função / variável está descrita por um comentário na imagem.

```cpp
// Classe NodeType representa os nós da árvore binária
class NodeType {
    public:
        /**
        * @brief Construtor padrão da classe NodeType.
        */
        NodeType();

        /**
        * @return O item no nó.
        */
        std::string getItem();

    private:
        std::string item; // O item armazenado no nó
        NodeType *esq;    // Ponteiro para o filho esquerdo
        NodeType *dir;    // Ponteiro para o filho direito

    friend class BinaryTree;
};
```

```cpp
class BinaryTree {
    public:
        /**
        * @brief Construtor padrão da classe BinaryTree que recebe uma expressão lógica.
        * @param expression A expressão lógica a ser usada na árvore.
        */
        BinaryTree(std::string expression);

        /**
        * @brief Destrutor da classe BinaryTree.
        */
        ~BinaryTree();

        /**
        * @return O ponteiro para a raiz da árvore.
        */
        NodeType* get_root();

        /**
        * @brief Insere um item na árvore.
        * @param item O item a ser inserido na árvore.
        */
        void insert(std::string item);

        /**
        * @brief Limpa a árvore, liberando a memória alocada.
        */
        void clear();

        /**
        * @brief Realiza uma travessia pós-ordem na árvore e calcula se a expressão é satisfazível.
        * @param p A raiz da árvore.
        * @return True se a expressão lógica dados quantificadores e valores fixos é satisfazível,
        *         False caso contrário.
        */
        bool posOrder(NodeType *p);

    private:
        /**
        * @brief Função auxiliar para inserção recursiva na árvore.
        * @param p A raiz da árvore.
        * @param item O item a ser inserido na árvore.
        */
        void InsertRecursive(NodeType* &p, std::string item);

        /**
        * @brief Função auxiliar para exclusão recursiva na árvore.
        * @param p A raiz da árvore.
        */
        void DeleteRecursive(NodeType* p);

        std::string expression_; // A expressão lógica avaliada na árvore
        NodeType *root; // Ponteiro para a raiz da árvore
};
```

2.3. Funções

Foram utilizadas 3 funções principais com o objetivo de avaliar as expressões lógicas. A primeira função converte a string de caracteres da expressão lógica para uma expressão pós-fixa, ou seja, uma expressão que organiza a string de forma que os caracteres e operadores fiquem na ordem certa de precedência(definida pela função auxiliar 'priority') ou, caso haja parênteses, a expressão menor de dentro do parênteses seja resolvida primeiro. Essa estratégia de conversão de infixo para pós-fixo é normalmente usada para resolver expressões matemáticas, mas foi adaptada para o problema proposto no trabalho.

A função 'evaluate' apenas itera pelo pós-fixo e com o auxílio de uma pilha realiza as operações e retorna o resultado.

```
/**
 * @brief Função para determinar a prioridade de um operador.
 * @param c O operador a ser avaliado.
 * @return A prioridade do operador (3 para '~', 2 para '&', 1 para '|', -1 para outros).
 */
int priority(char c);

/**
 * @brief Converte uma expressão infixada em uma expressão posfixada.
 * @param st A expressão infixada de entrada.
 * @param values Os valores associados aos operandos na expressão para conversão no posfixo.
 * @return A expressão posfixada resultante.
 */
std::string infix_to_postfix(std::string st, std::string values);

/**
 * @brief Avalia uma expressão lógica em formato posfixado.
 * @param st A expressão posfixada a ser avaliada.
 * @param values Os valores associados aos operandos na expressão.
 * @return O resultado da avaliação (true ou false).
 */
bool evaluate(std::string st, std::string values);
```

Além dessas funções para avaliar expressão, há duas outras auxiliares. Uma para a construção da árvore de valores e outra que junta os valores de dois de filhos de um vértice e substitui as letras da string por 'a', por 1 ou por 0 dependendo das strings de parâmetro.

```
/**
 * @brief Função para substituir letras 'a' ou 'e' por 0 ou 1.
 * @param s A string na qual a substituição será feita.
 * @param num O número binário para substituição.
 * @return A string resultante após a substituição.
 */
std::string change_letter_by_num(std::string s, char num);

/**
 * @brief Função que resume duas strings com base em uma lógica dos quantificadores.
 * @param s1 O filho esquerdo de um vértice.
 * @param s2 O filho direito de um vértice.
 * @return A string resultante após a aplicação da lógica.
 */
std::string resume_string(std::string s1, std::string s2);
```

3 Análise de Complexidade

- **Pilha - Complexidade de tempo:** como a pilha foi implementada por arranjo, todas as funções tem complexidade **O(1)**.
- **Pilha - Complexidade de espaço:** todas as funções da pilha tem complexidade espacial **O(1)**, já que não precisam de nenhum espaço auxiliar extra.
- **infixToPostfix - Complexidade de tempo:** para gerar a expressão pós-fixa é necessário iterar pela expressão inicial de tamanho n. A cada iteração, são feitas comparações e operações de complexidade **O(1)**, logo a complexidade de tempo total do algoritmo é **O(n)**.
- **infixToPostfix - Complexidade de espaço:** para a conversão da expressão, foi-se utilizada uma pilha de caracteres onde serão inseridos os valores e operadores da string, ou seja, na pilha serão inseridos os n caracteres da string e, por consequência, a complexidade de espaço da função é **O(n)**.
- **evaluate - Complexidade de tempo:** no início da função, primeiro se itera pela string de valores e verifica se há alguma letra ‘a’ para ser substituída, o que depende do tamanho da string(com certeza menor ou igual a ‘n’, já que não há como a string de valores ser maior que a expressão lógica). Logo após, chama-se a função infixToPosfix com complexidade O(n) e depois há a avaliação da expressão pós-fixa através de um loop pela string. Ou seja, a complexidade total é O(n) + O(n) + O(n) = O(3n) => **O(n)**.
- **evaluate - Complexidade de espaço:** como na função infixToPosfix, também foi utilizada uma pilha de inteiros auxiliar para avaliar a expressão, logo a complexidade espacial é de O(n).
- **Árvore - Complexidade de tempo:** como a árvore binária é balanceada, a complexidade de inserção na árvore é O(log n). A função que limpa a árvore tem complexidade O(V), sendo V o número de vértices da árvore, já que cada vértice é visitado e deletado uma vez. Por último, a função posOrder tem complexidade de tempo de O(V*(m + 2n + m)), pois a cada vértice é executada uma função que itera pela string de valores, depois são chamadas duas funções evaluate que tem complexidade O(n), ou seja O(n) + O(n) e, posteriormente, uma que também itera pela string de valores. Como m <= n em qualquer caso, podemos simplificar para O(V*4n) ou até O(V*n), sendo V o número de vértices da árvore binária e ‘n’ o tamanho da expressão lógica.
- **Árvore - Complexidade de espaço:** como a função evaluate é chamada duas vezes na em posOrder, a complexidade de espaço dessa função é O(2n) => O(n). O restante das funções possui complexidade(log n), ou seja, a altura da árvore.

É importante ressaltar a restrição imposta no trabalho na segunda parte, pois afeta completamente a complexidade do problema.

A restrição dita para resolver o problema é de que há um limite de 5 quantificadores no input de valores do programa. Era necessário um limite desse tipo, pois, como dito nos

tópicos acima, a complexidade da função posOrder é de O(V*n), sendo V o número de vértices. O número de vértices também pode ser traduzido para 2^m, sendo m o número de quantificadores no input, ou seja, a complexidade do problema é exponencial com relação ao número de quantificadores/letras nos valores e, por isso, se dá a restrição para que o tempo de execução não ficasse muito longo. Seguindo a mesma lógica dos vértices, a complexidade espacial também seria exponencial com relação ao número quantificadores na entrada do programa.

4 Estratégias de Robustez

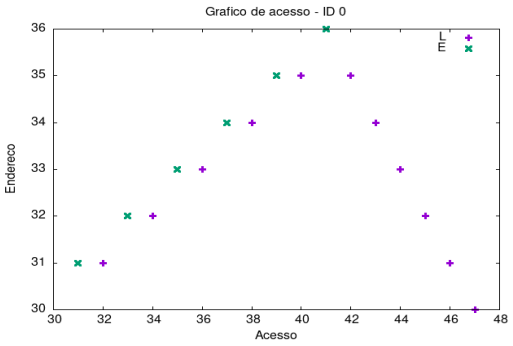
Para uma melhor robustez no código foram criadas classes de exceção na classe pilha para quando é retirado um item da pilha e ela está vazia ou quando se é inserido um item numa pilha que já está cheia, haja visto que a pilha foi implementada por arranjo e tem um tamanho máximo fixo. Para uso das exceções, foi-se utilizada a ferramenta try catch no arquivo main.cpp, como pode ser visto na imagem a seguir.

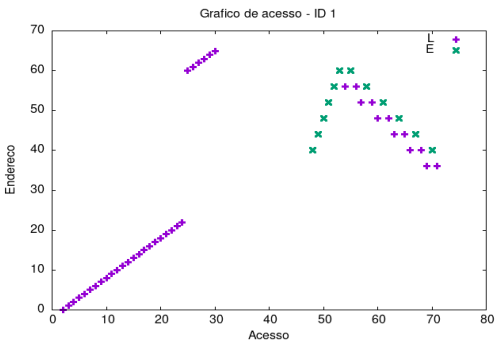
```
int main(int argc, char** argv){
    parse_args(argc, argv);

    if(opt == EVALUATEEXPRESSION){
        try{
            cout << evaluate(expression, values) << endl;
        } catch(stack_overflow_e e){
            cout << "A pilha está cheia!" << endl;
        } catch(empty_stack_e e){
            cout << "A pilha está vazia!" << endl;
        }
    }
    else if(opt == SATISFIABILITY){
        try{
            BinaryTree bt(expression);
            bt.insert(values);
            (bt.posOrder(bt.get_root())) ? cout << 1 << " " << bt.get_root()->getItem() << endl :
                                           cout << 0 << endl;
        } catch(stack_overflow_e e){
            cout << "A pilha está cheia!" << endl;
        } catch(empty_stack_e e){
            cout << "A pilha está vazia!" << endl;
        }
    }
    else cout << "Comando invalido!" << endl;

    return(0);
}
```

5 Análise Experimental

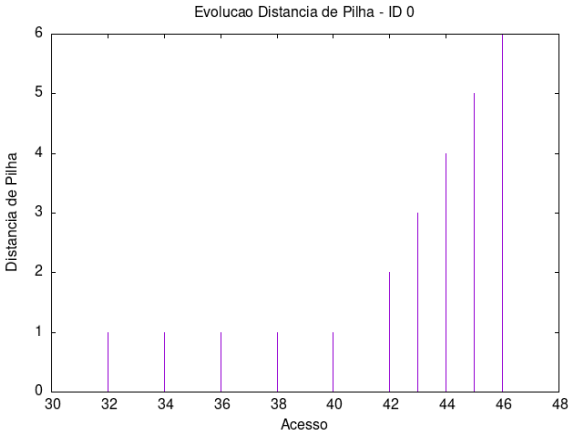
Através da biblioteca analisamem, foi possível analisar os acessos e distâncias de pilha das estruturas de dados pilha e árvore binária em ambas as partes do trabalho.

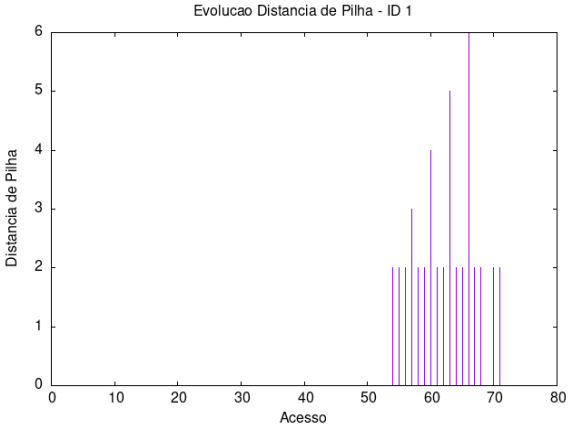
Parte 1: bin/tp1.out -a "0 | 1 | 2 & 3 & 4 & ~ 5" 001111 -p teste/evaluatelog.out -l



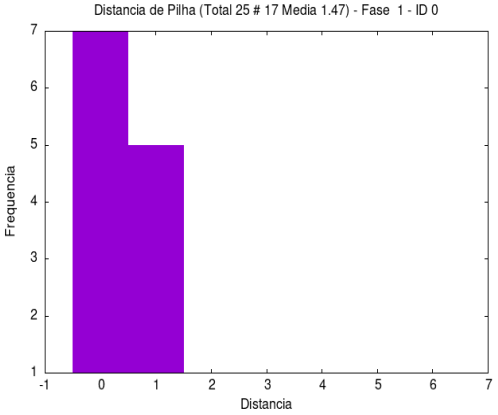


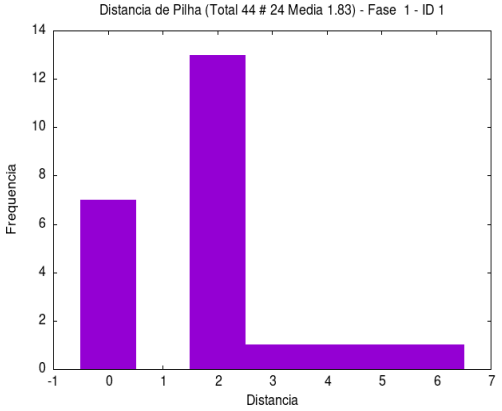
Nos gráficos de acesso de ID 0 e ID 1 (pilha de caracteres usada na função infixToPostfix e pilha de inteiros da função evaluate, respectivamente), fica claro o empilhamento e desempilhamento das pilhas utilizadas para avaliar as expressões. Ainda é possível perceber o padrão do segundo gráfico claramente, empilhando os valores e quando encontra um operador na expressão pós-fixa, desempilha dois e empilha um, desempilha dois e empilha um, e assim por diante.





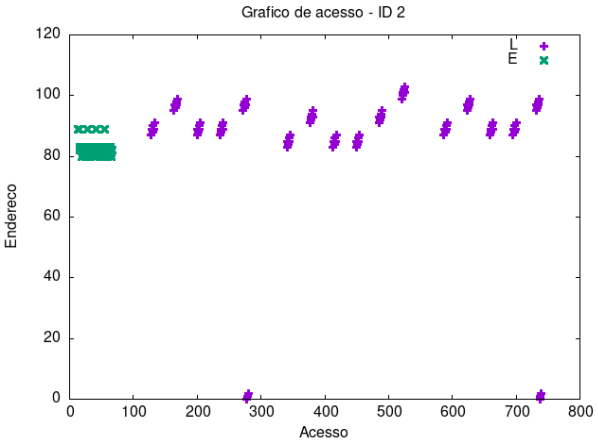
A evolução da distância de pilha descreve as diferenças de altura da pilha entre uma operação e a anterior. Por exemplo, levando os gráfico de acessos em conta, nota-se momentos em que a pilha tem 6 elementos empilhados um atrás do outro, e no gráfico de evolução de distância de pilha percebe-se um aumento linear a cada vez que esses valores são empilhados, aumentando de um em um até alcançar a distância máxima de 6.



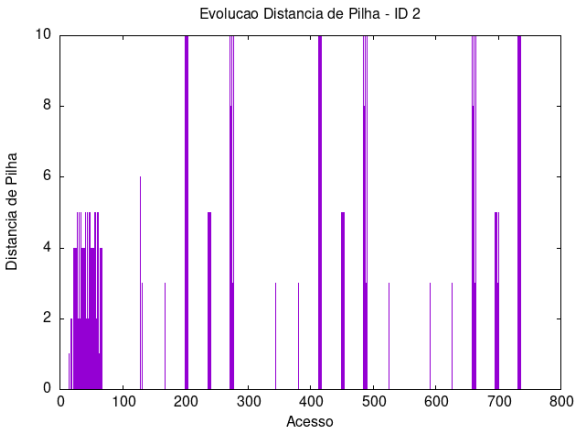


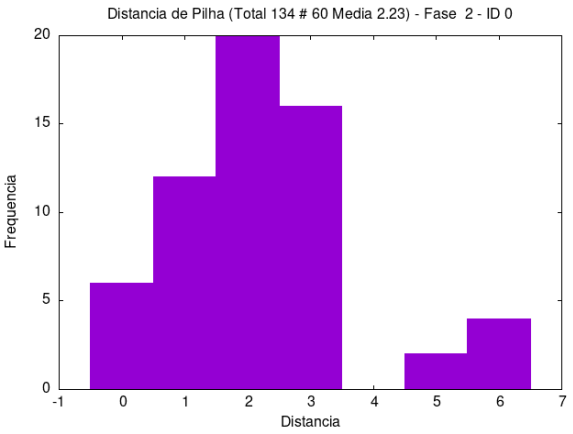
No histograma de distância de pilha, é mostrado quantas vezes aconteceu os empilhamentos/desempilhamentos seguidos nas pilhas, ou seja, quantas vezes, no gráfico de evolução de distância de pilha, ocorreram os acessos com uma diferença x na altura da pilha.

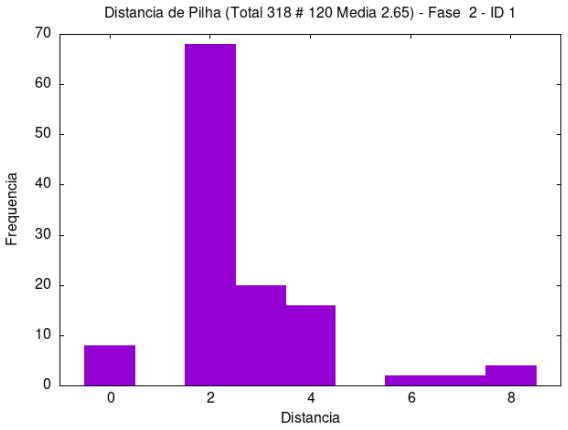
Parte 2: bin/tp1.out -s "0 | 1 | 2" aae -p teste/satisfiabilitylog.out -l

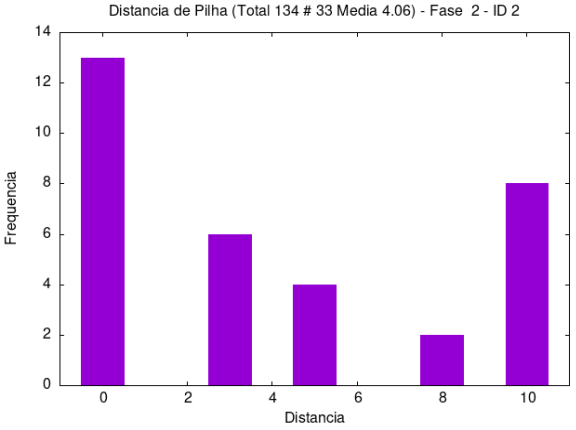


O gráfico de acesso da árvore binária mostra os vértices sendo criados no início do gráfico e, após isso, acessos em endereços em sequência representando as strings de cada vértice para a análise da satisfabilidade da expressão lógica com seus valores.









A primeira parte do gráfico representa a criação dos vértices da árvore e as três partes seguintes descrevem os acessos aos vértices, tanto para resolver o problema de satisfabilidade, quanto para destruir a árvore binária ao final do programa.

Como já citado anteriormente, o histograma de distância de pilha apenas organiza as quantidades de distância de pilha da primeira imagem e separando em fases, sendo a primeira a criação da árvore, a segunda o processo de resolução do problema dado o input do programa e a terceira fase a destruição de cada vértice da árvore binária.

6 Conclusão

A proposta do trabalho foi muito interessante para consolidar o aprendizado recente das estruturas de dados e das formas de analisar desempenho e complexidade de um código. Além disso, foi bom pensar em possíveis justificativas sobre as restrições impostas em cada parte do problema, como por exemplo a restrição de recursão na primeira parte e a restrição de 5 quantificadores na segunda parte.

Ademais, a construção do raciocínio para resolver, tanto o primeiro problema, como também e, principalmente, o segundo, foram de grande aprendizado ao longo do trabalho.

Bibliografia

http://www.vision.ime.usp.br/~pmiranda/mac122_2s14/aulas/aula13/aula13.html
https://algotree.org/algorithms/stack_based/infix_to_postfix/
https://leimao.github.io/blog/Argument-Parser-Getopt-C/
https://www.techiedelight.com/delete-given-binary-tree-iterative-recursive/

Instruções para compilação e execução

Para compilar o programa basta rodar **make all** na pasta raiz do projeto e, posteriormente rodar:

bin/tp1.out -opt “expressão” valores

Caso queira utilizar o avaliador de expressão troque ‘-opt’ por ‘-a’ e sugira a expressão(entre aspas) e valores que deseja substituir na expressão.

Exemplo:

```
 $ bin/tp1.out -a "0 | 1 | 2" 110
1
```

Caso queira avaliar a satisfabilidade da expressão com os quantificadores, substitua ‘-opt’ por ‘-s’ e a expressão(entre aspas) pela qual queira testar. Para os valores, use ‘a’ para o quantificador universal(∀) e ‘e’ para o quantificador existencial(∃).

Exemplo:

```
 $ bin/tp1.out -s "0 | 1 | 2" ae1
1 aa1
```