Trabalho 2

Teoria da Computação — Mestrado em Computação Aplicada

Fábio Pinto Monte
Ifes — Campus Serra — PPComp
2022/2

Solução 1

Nome do Quebra-Cabeças: Regex Quadrado

Descrição:

Este quebra-cabeças consiste em um quadrado de 7x7 células com alguns caracteres inseridos. O objetivo é encontrar uma expressão regular que preencha cada célula do quadrado, de modo que a expressão regular seja válida para todas as palavras na lista de palavras fornecida.

Lista de Palavras: regex, cross, word, square, puzzle, player.

Imagem do Quebra-Cabeças sem a Solução:

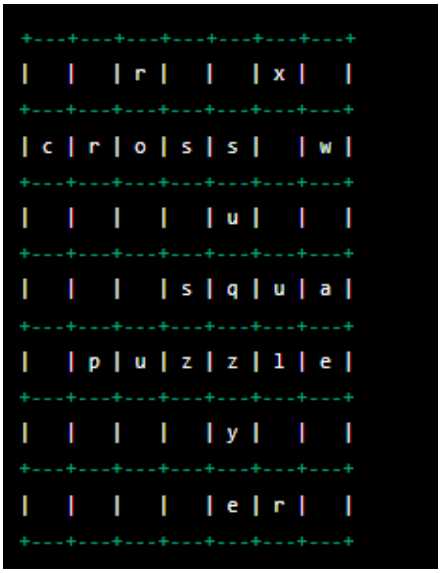
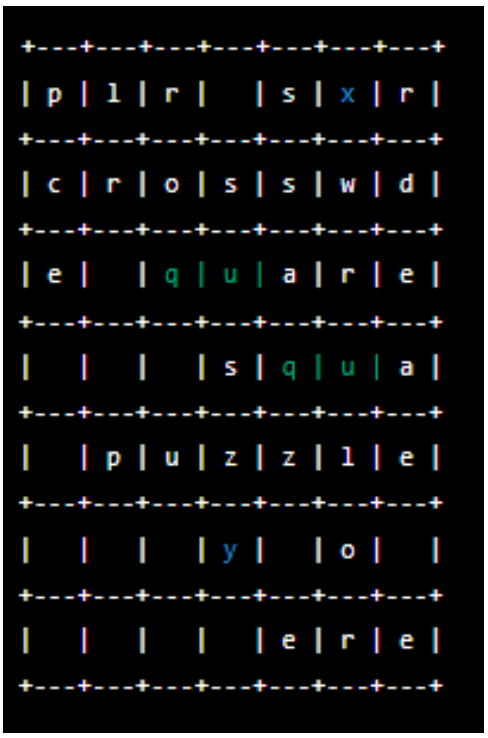


Imagem do Quebra-Cabeças com a Solução:



Expressão regular:

pr?l?s?e
cros?s?w?d
e?q?uar?e?
.?squa??
puzzle
.?y?.?
.?er?e

A avaliação do grau de dificuldade varia de pessoa para pessoa, mas acredito que esse quebra-cabeça tenha um nível médio de dificuldade. Ele pode ser resolvido em menos de 10 minutos por alguém com experiência em expressões regulares.

Solução 2

Parte 1:

Para implementar o programa, é necessário seguir os seguintes passos:

Parsear a expressão regular passada como parâmetro de linha de comando e transformá-la em uma árvore de sintaxe abstrata (AST).
Ler o texto a ser pesquisado a partir do arquivo ou da entrada padrão.
Percorrer o texto procurando por substrings que satisfaçam a expressão regular, utilizando a AST gerada anteriormente. Imprimir as *substrings* encontradas.

Parte 2:

Não foi feita

Solução 3

Parte 1

Para demonstrar que o problema de conjunto dominante simplificado é NP-completo, vamos realizar uma redução a partir do problema de cobertura de vértices (VERTEX COVER), que já é conhecido por ser NP-completo.

O problema de cobertura de vértices é definido da seguinte maneira: Dado um grafo $G = \langle V, E \rangle$ e um inteiro k, determinar se existe um conjunto $V' \subseteq V$ com $|V'| \leq k$ tal que cada aresta em E tenha pelo menos um dos seus extremos em V'.

Para realizar a redução, vamos mostrar que o problema de conjunto dominante simplificado pode ser utilizado para resolver o problema de cobertura de vértices. Dado um grafo $G = \langle V, E \rangle$ e um inteiro k, vamos construir um novo grafo $G' = \langle V', E' \rangle$ da seguinte maneira:

1. Adicione um vértice novo v_i para cada aresta (u, w) em E.
2. Adicione uma aresta (v_i, v_j) em E' para cada par de vértices v_i e v_j que correspondam às arestas que compartilham um mesmo vértice em G.

O número de vértices em G' é $2|E|$, pois há um vértice novo para cada aresta em G. Além disso, cada vértice novo está conectado a um conjunto de vértices em G. Assim, se um vértice v_i em G' é adicionado ao conjunto dominante D, então pelo menos um dos vértices que ele está conectado em G também deve estar em D. Dessa forma, o conjunto D em G' corresponde a uma cobertura de vértices em G.

Portanto, podemos utilizar um algoritmo que resolva o problema de conjunto dominante simplificado em G' para resolver o problema de cobertura de vértices em G. Como o

problema de cobertura de vértices é NP-completo, concluímos que o problema de conjunto dominante simplificado também é NP-completo.

Parte 2

Não foi feita

Solução 4

Parte 1

Para provar que o problema do recrutamento é NP-completo, é necessário mostrar que ele pertence à classe NP e que ele é pelo menos tão difícil quanto o problema da cobertura por conjuntos (Set Cover), que é conhecido por ser NP-completo.

Primeiro, para mostrar que o problema do recrutamento pertence à classe NP, é possível construir um certificado para a solução do problema, que seria a lista dos índices dos empregados escolhidos para a equipe. Dado esse certificado, é possível verificar em tempo polinomial se ele cobre todas as competências necessárias.

Agora, para mostrar que o problema do recrutamento é pelo menos tão difícil quanto o problema da cobertura por conjuntos, é possível fazer uma redução polinomial do problema da cobertura por conjuntos para o problema do recrutamento. Dado um conjunto S e uma coleção C de subconjuntos de S, o problema da cobertura por conjuntos consiste em encontrar o menor subconjunto de C que cobre todos os elementos de S.

Para reduzir esse problema para o problema do recrutamento, é possível criar uma lista de candidatos, onde cada candidato é associado a um subconjunto de C. Além disso, cada elemento de S é associado a uma competência, e o número total de competências é igual a |S|. Dessa forma, o problema do recrutamento consiste em encontrar um subconjunto de k candidatos que possa cobrir todas as competências (ou seja, todos os elementos de S). Se for possível resolver o problema do recrutamento em tempo polinomial, então também é possível resolver o problema da cobertura por conjuntos em tempo polinomial, o que mostra que o problema do recrutamento é NP-completo.

Parte 2

A ideia da implementação é escolher o candidato que possui a maior interseção com as competências restantes e adicioná-lo à equipe. Se a interseção for vazia, não há solução

possível e a função retorna None. Se a equipe estiver completa, a função retorna a lista dos índices dos empregados escolhidos.

Parte 3
Uma possível alteração para tornar o problema do recrutamento mais eficiente é considerar uma abordagem gulosa, em que sempre escolhe o candidato que cobre o maior número de competências ainda não cobertas

Solução 5

Parte 1
Problema de nível médio: Passageiros de trem

Vamos começar com a definição do problema: queremos verificar se as medições do trem são consistentes, ou seja, se em todas as estações o número de pessoas no trem não ultrapassou a capacidade total e nem ficou abaixo, e nenhum passageiro esperou em vão. Além disso, o trem deve começar e terminar a viagem vazio, e em particular, os passageiros não devem esperar pelo trem na última estação.

Para resolver o problema, podemos manter uma variável passageiros que representa o número de passageiros no trem a cada estação. Inicialmente, passageiros é zero. Em seguida, percorremos cada estação, atualizando passageiros com o número de passageiros que entrou e subtraindo o número de passageiros que saiu. Se em algum momento passageiros ficar maior que a capacidade total do trem ou menor que zero, a resposta é "impossível". Além disso, se em alguma estação houver passageiros esperando e passageiros não estiver no máximo, a resposta também é "impossível". Por fim, se passageiros for diferente de zero na última estação, a resposta é "impossível".

O algoritmo lê a capacidade e o número de estações do trem na primeira linha. Em seguida, percorre as n_estacoes estações, lendo o número de passageiros que saíram, entraram e esperaram em cada uma delas. Se em algum momento a resposta for "impossível", o programa interrompe o loop com o comando break. Se o loop terminar sem encontrar problemas, a resposta é "possível" se passageiros for igual a zero, ou "impossível" caso contrário.

Parte 2

Problema de nível difícil: Subsequência crescente mais longa

O código utiliza a função lis para encontrar a subsequência crescente mais longa em uma sequência de números inteiros. Essa função usa uma técnica chamada "Algoritmo da Subsequência Crescente Mais Longa" (LIS - Longest Increasing Subsequence) que tem complexidade de tempo O(n log n) e espaço O(n), onde n é o comprimento da sequência.

O código lê a entrada em um loop que termina quando não há mais casos de teste a serem lidos. Para cada caso de teste, ele lê o comprimento da sequência e os números inteiros que a compõem. Em seguida, ele chama a função lis para encontrar a subsequência crescente mais longa e imprime o comprimento da subsequência e os índices dos elementos que a compõem.

O código utiliza a função bisect_left do módulo bisect para encontrar o índice em que um elemento deve ser inserido em uma sequência ordenada. Isso é usado para atualizar a sequência de índices seq_idx que mantém o menor final para cada subsequência crescente de comprimento i+1. A lista prev_idx é usada para manter um ponteiro para o índice anterior na subsequência mais longa que termina no índice i.

A função lis retorna uma tupla com o comprimento da subsequência crescente mais longa e um iterável com os elementos que a compõem em ordem reversa. O loop final no código usa essa iterável para reconstruir a subsequência em ordem crescente e encontrar seus índices na sequência original.