MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

Laboratory work 1:
Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-223
Vieru Mihai

Verified:
asist. univ.
Cretu Dumitru

Chișinău, 2024

Content

IMPLEMENTATION

ProblemB:

In order to solve problem B, I utilized a recursive backtracking approach. Because my grammar has only one valid string, I had to utilize this approach in order to get 5 strings. My method can generate all of the strings until n recursions, or until the stack of the computer manages.

Also because of its nature it works with any grammar.

Here is the main method:

Implementation:

```
def getStrings(self):
    result_strs = set()
    size = 5
    def iter(grammar_str, result_str, visited, NT):
        # This is used to break out of the recursion when the reuired size
         ↪  of strings is found
        if len(result_strs) == size:
            return

        for chars in grammar_str:
            for char in chars:
                if self.parsed_grammar.get(char):
                    # this visited set is used in order to see, how many
                     ↪  times we have visited a node
                    visited.setdefault(char, 0)
                    visited[char] += 1
                    # Here you can specify how many recursive cycles can
                     ↪  be done
                    if visited[char] < 3:
                        iter(self.parsed_grammar[char], result_str.copy(),
                         ↪  visited.copy(), char)
                    # here we are removing the previous attempt, aka
                     ↪  backtracking
                    result_str.pop()
```

```
                else:
                    result_str.append(char)
            if len(result_strs) == size:
                return
            # Checks if we reached the final state
            if NT in self.end_symbols:
                result_strs.add("".join(result_str))

        iter(self.parsed_grammar[self.start_symbol], [], {}, self.start_symbol)
        return result_strs
```

Problem C:

Here I just make a hash map with all the nodes and the nodes they connect to and use the matplotlib library to plot them, some edge cases I had to solve was a cycle conection so a conection overwrote each other, so I had to combine them, and that self cycles did not represent their terminals. But these problems have been fixed.

Now it works with all grammars.

Implementation:

```
def generateGraph(self):
    G = nx.DiGraph()

    edge_labels = {}
    self_loop = {}
    # Go through all conections of the hash map
    for node, neighbors in self.data.items():
        for neighbor in neighbors:
            transition = neighbor[0]
            target_node = neighbor[1:]
            G.add_edge(node, target_node)
            # check for self loops, if any store to display them later
            if node == target_node:
                if node in self_loop:
```

```
                    self_loop[node] += f", {transition}"
                else:
                    self_loop[node] = transition
            # if a cycle connection append them togheter
            if (target_node, node) in edge_labels:
                edge_labels[(target_node, node)] += f", {transition}"
            else:
                edge_labels[(node, target_node)] = transition

    pos = nx.spring_layout(G, k=0.5)
    nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=1500,
     ↪  edge_color='gray', arrowsize=20)

    # Display the circles around the end symbols
    for symbol in self.end_symbols:
        circle_radius = 0.25
        circle_center = pos[symbol]
        circle_patch = plt.Circle(circle_center, circle_radius,
         ↪  color='red', fill=False)
        plt.gca().add_patch(circle_patch)

    # display the self loop values
    for node, transition in self_loop.items():
        pos_labels = pos.copy()
        pos_labels[node][1] += 0.10
        nx.draw_networkx_labels(G, pos_labels, labels={node: transition},
         ↪  font_color='black')

    # display the initial node
    pos_labels = pos.copy()
    pos_labels[self.start_symbol][0] -= 0.13
    nx.draw_networkx_labels(G, pos_labels, labels={self.start_symbol:
     ↪  '->'}, font_color='black', font_weight='bold')
```

```
        # draw the graph
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
        plt.show()
```

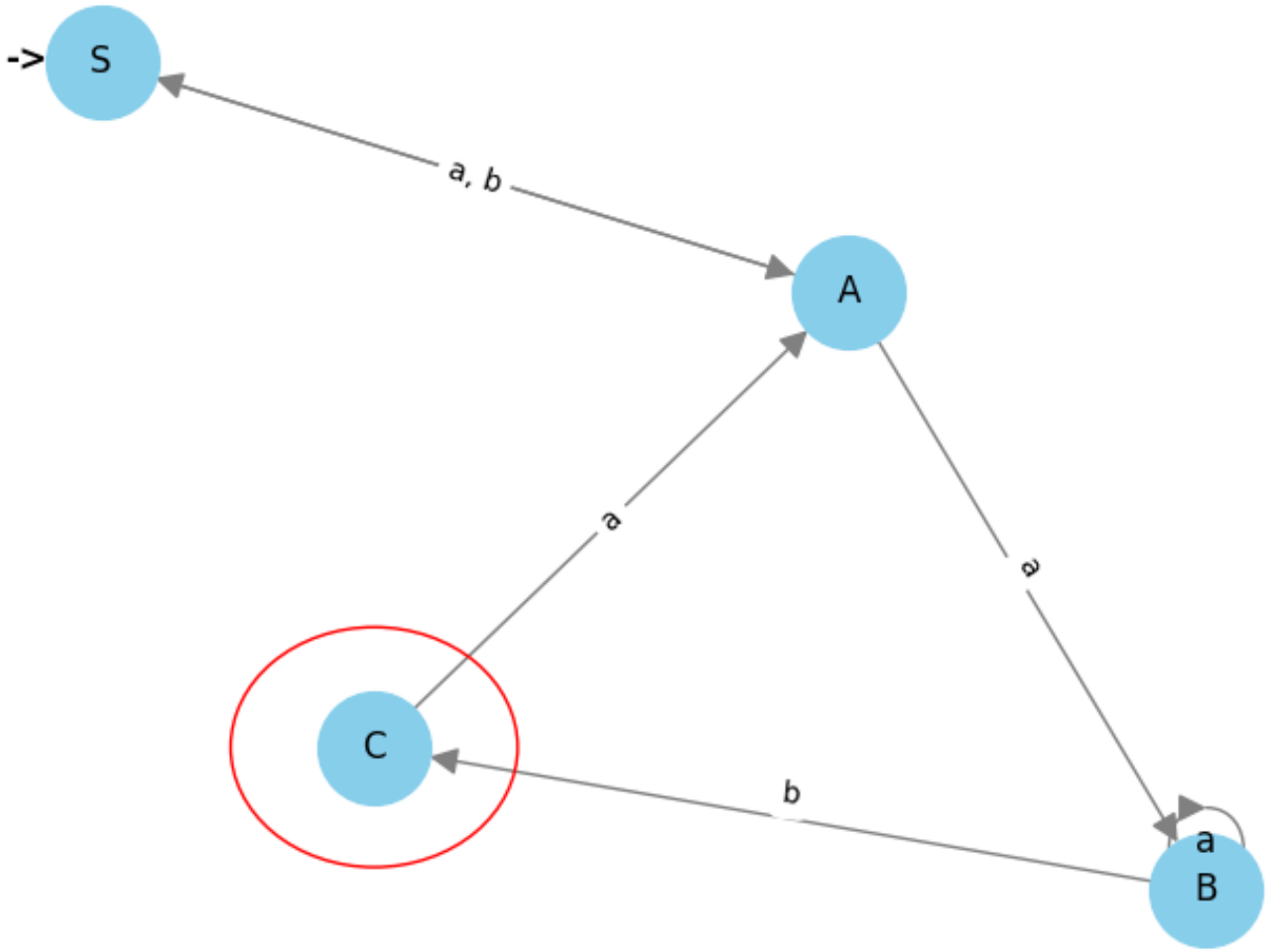

Figure .1 - ProblemC Example

Problem D:

Here, I used a recursive backtracking approach again. We check every single choice we have from the first one. If none of them are equal to the current index of the word we need to create, that means the word cannot be created with this grammar. However, if one of them is equal to the value at the current index of the needed string, we check that path. We increase the index of the string, then check if any of the next possibilities have the current index value of the needed string, and so on. It checks all possibilities, and if none are found, it's returned as False. If at least one is True, we return True.

Implementation:

```
def checkStr(self, check_str):
    def iter(grammar_str, i, path_str, NT):
        # if we went too far and now the string we can create is bigger
        ↪ than the needed string
        if i > len(check_str) - 1:
            return path_str == check_str

        for chars in grammar_str:
            # This is only for the possibility that we reached a terminal
            ↪ state and we cannot do any more recursion
            if len(chars) == 1:
                if chars == check_str[i]:
                    return iter(self.parsed_grammar[NT], i + 1, path_str +
                    ↪ chars, NT)
            #This is the general state
            else:
                # Here we check if the current possibility is equal to the
                ↪ value of the current index of the string, if yes
                ↪ explore that sub tree
                if chars[0] == check_str[i]:
                    if iter(self.parsed_grammar[chars[1]], i + 1, path_str
                    ↪ + chars[0], chars[1]):
                        return True
        return False

    return iter(self.parsed_grammar[self.start_symbol], 0, "",
    ↪ self.start_symbol)
```

CONCLUSION

Recursion is useful.

PS: I don't know if there is a need for a conclusion.