Scrivo questo documento per condividere alcune scelte progettuali riguardo alla soluzione proposta.

Formalmente, un automa finito deterministico è una quintupla $M = (\Sigma, Q, q_0, Q_f, \delta)$ dove:

- Σ è l'alfabeto di input
- Q è l'insieme finito i cui elementi sono detti *stati dell'automa*
- q_0 è l'elemento speciale detto *stato iniziale*
- $Q_f \subseteq Q$ è l'insieme degli stati finali.
- δ è la funzione che determina le transizioni di stato; mappa coppie $\langle stato, simbolo \rangle$ in stati: $\delta : Q \times \Sigma \to Q$

Il programma elabora sia automi finiti deterministici che non deterministici.
Ho valutato diverse opzioni che consentissero di implementare entrambi i tipi di automi:
1. La definizione di due tipologie di automi tramite *struct*
2. La definizione di due tipologie di automi tramite *class*
3. La definizione di una *class* generica, da specializzare in NDFA e DFA.

```
template <typename State, typename Symbol, typename
TransitionType>
class Automaton {

    private:
        State initialState;
        State finalStates;
        set<Symbol> alphabet;
        TransitionType transitions;
        vector<State> allStates;

    public:
        Automaton () {}

    void setInitialState(const State& initialState) {
        this->initialState = initialState;
    }

    void setFinalStates(const State& finalStates) {
        this->finalStates = finalStates;
    }

    void setAlphabet(const set<Symbol>& alphabet) {
        this->alphabet = alphabet;
    }

    void setTransitions (const TransitionType& transitions) {
        this->transitions = transitions;
    }

    const TransitionType& getTransitions() const {
        return transitions;
    }

    const State& getStartState () const {
        return initialState;
    }

    const State& getFinalStates() const {
        return finalStates;
    }

    const set<Symbol>& getAlphabet() const {
        return alphabet;
    }

    void addState(const State& state) {
        if (find(allStates.begin(), allStates.end(), state)
== allStates.end())
            allStates.push_back(state);
    }

    const vector<State>& getAllStates () const {
        return allStates;
    }

    void printAll () const {
        // default automaton print
    }
```

Seguendo il suggerimento di "esercitarsi sulla *generic programming*", ho provato ad definire una *class* unica, sfruttando i *template*.

La *class* Automaton utilizza tre parametri di template:
1. **State**: rappresenta il tipo di dato per rappresentare gli stati dell'automa.
2. **Symbol**: rappresenta il tipo di dato per rappresentare i simboli dell'alfabeto dell'automa.
3. **TransitionType**: rappresenta il tipo di transizioni dell'automa

I tre tipi sono stati utili per differenziare le funzioni dedicate all'elaborazione di dati degli automi finiti non deterministici e degli automi finiti deterministici.

Negli automi finiti non deterministici:

`Automaton<int, char, multimap<pair<int, char>, int>>`
• Gli stati vengono indicati con i numeri interi: *State* viene specializzato con il tipo *int*.
• Lo stato iniziale (o finale) abbiamo deciso essere indicato sempre e solo da uno stato: *State* specializzato *int* è corretto.
• Le transizioni vengono rappresentati come *multimap*: questo tipo di automi comprende le ϵ-transizioni; si rende necessaria una struttura dati che accetti istanze con duplicati di chiave.

Negli automi finiti deterministici:

`Automaton<set<int>, char, map<pair<set<int>, char>, set<int>>>`
• Gli stati vengono indicati con gruppi di numeri interi: *State* viene specializzato con il tipo *set<int>*.
• Lo stato iniziale (o finale) è indicato anch'esso da un gruppo di numeri interi: *State* specializzato *int* è corretto.
• Le transizioni vengono rappresentati come *map*: questo tipo di automi non comprende le ϵ-transizioni; è necessaria l'adozione di una struttura dati che escluda istanze con duplicati di chiave.

Ho deciso di adottare tipi parametrizzati per esercitarmi sulla *generic programming* e anche perché, in un futuro, la stessa *class* potrà essere utilizzata con altri tipi di dato.

Per esempio, nell'esercizio proposto, *Symbol* è sempre di tipo *char*: l'alfabeto è sempre composto da caratteri singoli; in un futuro, oggetti della stessa classe potranno essere eventualmente istanziati per automi con alfabeti diversi.

Condivido un appunto ulteriore per *vector<State>*.
Inizialmente, la mia soluzione non prevedeva un contenitore per memorizzare l'elenco degli stati dell'automa.
Anche se introduce ridondanze, ho pensato che contenere un elenco di stati *ready to use* fosse una decisione utile.
Ho scelto di utilizzare *std::vector* per non complicare l'utilizzo dei *set*: questi non garantiscono l'ordine di inserimento degli elementi.
allStates viene popolato durante la costruzione dell'automa e viene utilizzato nelle procedure di stampa dei dati.

In generale, ho deciso di inserire i *template* e usare questo tipo di *class* principalmente per accrescere e migliorare la mia preparazione.

Elia Pitzalis matr. 152541
284660@studenti.unimore.it