بسمه تعالی

نظریه زبان‌ها و اتوماتا — نیمسال اول ۹۸-۹۹
گزارش پروژه — مهرسا پوریا (۹۵۱۰۱۲۴۷)

جعبه ابزار ماشین‌های حالت متناهی

نکته آغازی : لطفا در ورودی دادن به کد در کنار حالتهای انتهای یک خط اسپیس اضافی نگذارید زیرا عملیات خواندن با مشکل روبه رو میشود؛ همچنین چون برای الگوریتم تبدیل به دی اف ای از الگوریتم کتاب استفاده شده و تعداد حالات آن دو به توان تعداد حالات ان اف ای ورودی است؛ برای ساده سازی که جدول به ابعاد تعداد متغیرها زیاد شود ممکن است با ارور حافظه رو به رو شوید. (برای تبدیل الگوریتم‌های بهینه تر است اما چون هدف یادگیری مفاهیم درس بود همان الگوریتم کتاب پیاده شد که این مشکل را دارد.)

۱ تبدیل عبارت منظم به **NFA** و برعکس

۱.۱ تبدیل **NFA** به عبارت منظم

برای اینکار از الگوریتم GNFA استفاده می‌کنیم. یعنی در ابتدا دو حالت شروع و پایان اضافه می‌کنیم؛ حالت شروع جدید با اپسیلون به حالت شروع NFA می رود و از حالتهای پایان NFA با اپسیلون به حالت پایان جدید می‌رویم. سپس تا زمانی که فقط دو حالت افزوده باقیمانده باشد همه‌ی حالت‌های غیر این دو را حذف میکنیم و یال‌ها را به صورت زیر آپدیت میکنیم؛ عبارت منظمی که روی یال آخرین مرحله از حالت شروع افزوده به حالت نهایی افزوده وجود دارد همان عبارت منظم مورد نظر ما خواهد بود.
آپدیت کردن یال ها نیز به این صورت است که در همه‌ی مسیرهای ۳تایی که حالت حذف شده نقطه‌ی میانی آن است، یال بین دو حالت ابتدا و انتهای این مسیر را با اجتماع گرفتن با یال قبلی و عبارت منظم مسیر حذفی که معادل الحاق به ترتیب پیمایش (ستاره گرفتن در صورت وجود طوقه برای حالت محذوف) است؛ آپدیت میکنیم. این الگوریتم برگرفته از کتاب است و خلاصه آن در شکل زیر آمده است.

CONVERT(G):

1. Let k be the number of states of G.

2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.

3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where

$$Q' = Q - \{q_{\text{rip}}\},$$

and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.

4. Compute CONVERT(G') and return this value.

شکل ۱: الگوریتم GNFA

۱.۱.۱ توضیح کد

هم در این قسمت و هم در بخش بعد یعنی تبدیل NFA به DFA مینیمم برای خواندن و ذخیره اطلاعات ورودی یک ابجکت از کلاس NFA که تعریف کرده‌ایم می‌سازیم و با صدا زدن تابع getInput آن، به field های آن مقدار می‌دهیم؛ مقادیری که بسته به خروجی مد نظر در الگوریتم‌ها مورد استفاده قرار می‌گیرند. متغیرهای کلاس NFA به صورت زیر هستند.

```
string startState; // name of NFA's start state
vector<string> FinalStates; // name of final states
vector<string> alphabet; // alpahbets
vector<string> states; // name of states
map<string, int> state2num; // we store integer number related to each state name here
// 2 next used for NFA to RegEx
map<int, string>* delta; // delta[i] stores (j,exp) which there is an edge from i to j by exp
map<int, string>* deltaInv; // deltaInv[i] stores (j,exp) which there is an edge from j to i by exp
// next used for NFA to minDFA
multimap<string, int>* deltaTo; // deltaTo[i] stores all (s,j) : there is an edge from i to j by s
int nF = 0, nS = 0, nSt = 0; // nF : number of Final States, nS : size of alphabet, nSt : number of states
```

شکل ۲: متغیرهای کلاس NFA

نکته پیاده سازی : برای سادگی در خواندن ورودی دو حالت ابتدا و انتهای اضافی برای GNFA را به متغیرها در همان ابتدا اضافه میکنیم؛ اگر بعدا از چنین متغیری در بخش مربوط به DFA استفاده کنیم، کافی است اعداد نسبت داده شده را یکی شیفت دهیم، همچنین با توجه به تفاوت الگوریتم‌ها و برای سادگی هریک توابع انتقال را در داده ساختارهای مختلف نگه می‌داریم که در شکل بالا نیز در کامنت‌ها به آن اشاره شده‌است.
برای رسیدن به عبارت منظم حال تابع toRegEx را صدا میزنیم که توسط تابع deleteState همه‌ی حالتها غیر حالت شروع و آخر افزوده را حذف میکنیم؛ یالها را با توجه به الگوریتم گفته‌شده در ابتدا آپدیت کرده و در نهایت یال بین دو حالت مانده را به عنوان عبارت منظم مورد نظر باز می‌گردانیم.

```
void toRegEX() {
    for (int i = 1; i < nSt - 1; i++) {
        deleteState(i);
    }
    cout << delta[0].find( x: nSt-1)->second;
}
```

شکل ۳: تبدیل به عبارت منظم

تابع deleteState مطابق الگوریتم گفته شده یالها را آپدیت میکند؛ با توجه به پیاده‌سازی هم delta و هم deltaInv که در کامنت‌های شکل ۲ کاربردهایشان گفته‌شده است؛ را به‌روز رسانی می‌کنیم.

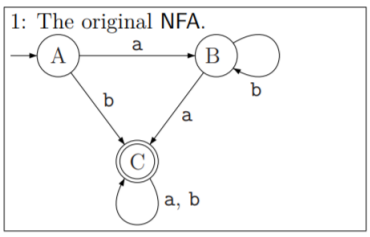
۲.۱.۱ بررسی مثال

برای مثال داده‌شده در متن توضیح پروژه خروجی به صورت زیر است که صحیح است.

```
NFA
q1
q6
a b
q1 q2 q3 q4 q5 q6
q3 - q2
q2 q4 -
- q3 q4
q5 - -
- q6 -
- - q1
RegEx
((a)*b|a(b)*)ab(((a)*b|a(b)*)ab)*
```

شکل ۴: مثال ۱

برای شکل زیر نیز، عبارت منظم محاسبه شده؛ که جوابش صحیح شد.

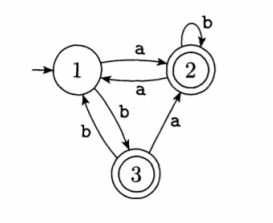


شکل ۵: شکل مثال ۲

```
NFA
A
C
a b
A B C
B C -
C B -
C C -
RegEx
(b|a(b)*a)((a|b))*
```

شکل ۶: ورودی خروجی مثال ۲

برای مثال زیر نیز که از کتاب است جواب با پاسخ کتاب معادل است.



شکل ۷: شکل مثال ۳

```
NFA
q1
q2 q3
a b
q1 q2 q3
q2 q3 -
q1 q2 -
q2 q1 -
RegEx
(a((b|aa))*|(b|a((b|aa))*ab)((bb|(a|ba)((b|aa))*ab))*(ε|(a|ba)((b|aa))*))
```

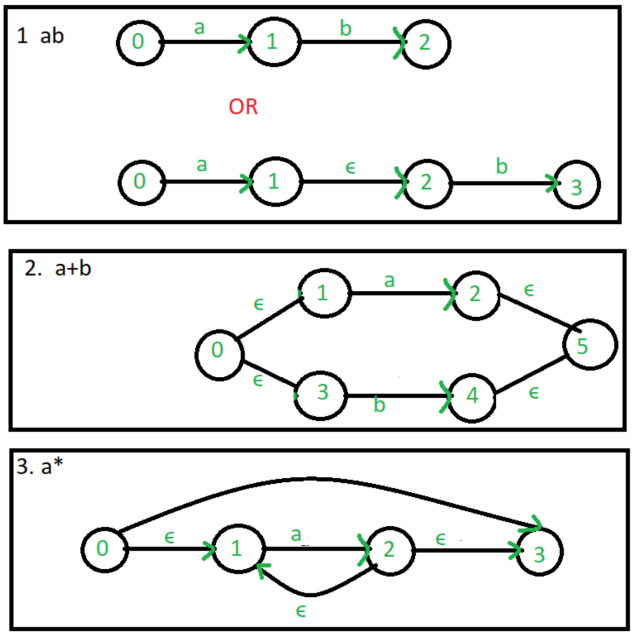
شکل ۸: ورودی خروجی مثال ۳

۲.۱ تبدیل عبارت منظم به **NFA**

میدانیم الحاق؛ اجتماع و ستاره را میتوانیم توسط NFA بسازیم؛ و چون هر عبارت منظم از این اعمال به دست می آید میتوانیم هر عبارت منظم را هم توسط NFA بسازیم.

۱.۲.۱ پیاده‌سازی

برای پیاده‌سازی بلوک‌های پایه‌ای را هرجا که لازم بود؛ مطابق شکل زیر می‌سازیم. دو استک داریم به نام‌های s1 و s2 که برای هر عبارتی که به NFA تبدیل می‌کنیم ابتدا و انتهای آن را به ترتیب در این s1 و s2 ذخیره می‌کنیم. یک پشته به نام operators داریم که پرانتز باز و | را به آن وارد میکنیم (در هنگام پرانتز باز -1 را به s1و s2 نیز اضافه میکنیم) و هنگامی که پرانتز بسته دیدیم بسته به اینکه بعد آن ستاره هست یا نه ابتدا داخل آن را ساده میکنیم یعنی تا جایی که به -1 برسیم ابتدا و انتهای قبلی را از s1 و s2 برمیداریم و بنا بر اپراتوری که در آن زمان با آن کار میکنیم بلوک های جدید میسازیم. و ابتدا و انتهای جدید را به پشته وارد می‌کنیم، همین روند را ادامه می‌دهیم تا زمانی که عبارت تمام شود. در همین میان توابع انتقال را در هر مرحله اپدیت میکنیم؛ در نهایت هم متغیر باقی مانده در s1 متغیر شروع و در s2 متغیر پایان است. این کارها توسط تابع toNFA انجام شده‌اند که خروجی آن یک آبجکت از کلاس تعریف شده NFA است. (بنابراین برای تبدیل عبارت منظم به DFA راحت میتوانیم از متد toDFA کلاس ان اف ای استفاده کرد.)



شکل ۹: حالتهای ساخت بلوک اساسی؛ مزیت این نوع ساختن این است که ابتدا و انتها را به راحتی میتوان در استک قرار داد. در حالت الحاق از حالت بدون یال اپسیلون وقتی استفاده میکنیم که حروف الفبا پشت هم آمده باشند و از با یال اپسیلون وقتی که بخواهیم اعضای پشته را پشت هم بگذاریم استفاده می‌کنیم. عکس برگرفته از سایت گیکز فور گیکز است.

۲.۲.۱ بررسی مثال

یک مثال ساده پیاده شده است که صحیح است:

```
RegEx
(a|b)
NFA
q7
q8
a b
q2 q3 q5 q6 q7 q8
q3 - -
- - q8
- q6 -
- - q8
- - q2,q5
- - -
```

شکل ۱۰: مثال ۱

همچنین برای همین مثال ساده خروجی دی اف ای که یکتاست نیز به صورت زیر است که قابل قبول است. (با توجه به مذکورات نکته آغازی در اینجا هم اگر تعداد حالات ان اف ای زیاد شود چون الگوریتم مربوط به تبدیل ان اف ای به دی اف ای کتاب و بعدش مینیمم کردن حافظه زیادی میخواهد ارور حافظه رخ می‌دهد.)

```
RegEx
(a|b)
DFA
q1
q2
a b
q0 q1 q2
q0 q0
q2 q2
q0 q0
```

شکل ۱۱: تکمیلی مثال ۱

۲ تبدیل NFA به DFA مینیمم

برای این قسمت ابتدا NFA را به یک DFA مطابق زیر تبدیل می‌کنیم.

$$\begin{cases} \hat{Q} = P(Q) = 2^Q \\ \hat{q_0} = E(q_0) \\ \hat{F} = \{R \in \hat{Q} | \text{R contains an accept state of NFA} \\ \delta(\hat{R}, a) = \{q \in Q | q \in E(\delta(r, a)) \text{for some r} \in R\} \end{cases} \quad (1)$$

که در آن $E(.)$ از رابطه زیر به دست می‌آید.

$$E(R) = \{q | \text{q can be reached from R by traveling along 0 or more } \epsilon \text{ arrows}\} \quad (2)$$

سپس با استفاده از الگوریتم آمده در فایل ضمیمه آن را به DFA مینیمم تبدیل می‌کنیم. یعنی ابتدا حالاتی را که از متغییر شروع قابل دسترسی نیستند حذف و سپس الگوریتم شکل زیر را اجرا میکنیم.

A Recursive Definition of S_M

The set S_M can be defined as follows:

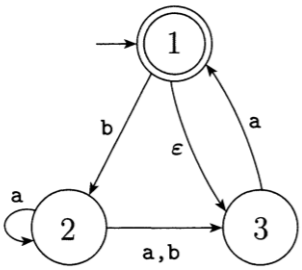
1. For every pair (p, q) with $p \neq q$, if exactly one of the two states is in A, $(p, q) \in S_M$.
2. For every pair (r, s) of distinct states, if there is a symbol $\sigma \in \Sigma$ such that the pair $(\delta(r, \sigma), \delta(s, \sigma))$ is in S_M, then $(r, s) \in S_M$.

شکل ۱۲: الگوریتم مینیمم کردن DFA

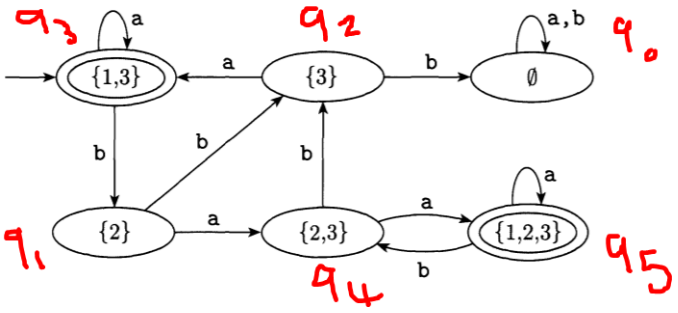
تابع toDFA کلاس NFA این کار را انجام میدهد. ابتدا E را حساب و بعد دی اف ای را طبق کتاب میسازد و سپس طبق فایل ضمیمه مینیمم میکند.

۱.۲ مثال

برای مثال کتاب نتایج درست شد.



شکل ۱۳: مثال ۱ : ان اف ای اولیه



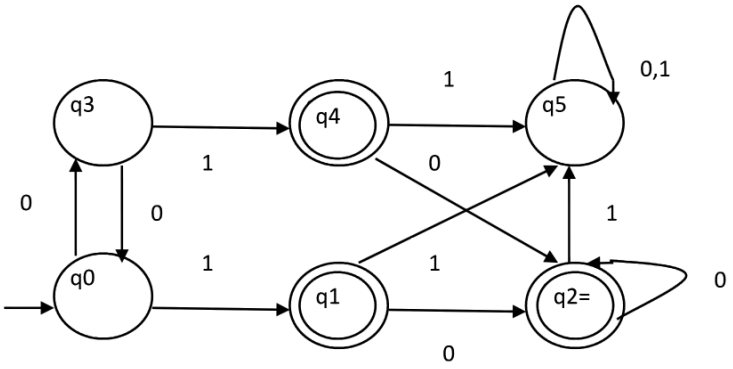
شکل ۱۴: مثال ۱: جواب کتاب

```
NFA
q1
q1
a b
q1 q2 q3
- q2 q3
q2,q3 q3 -
q1 - -
DFA
q3
q3 q5
a b
q0 q1 q2 q3 q4 q5
q0 q0
q4 q2
q3 q0
q3 q1
q5 q2
q5 q4
```

شکل ۱۵: مثال ۱ : خروجی و ورودی کد

۲.۲ مثال جالب

دی اف ای زیر که یک ان اف ای است را در نظر بگیرید.



شکل ۱۶: ماشین اولیه

حال اگر عبارت منظم آن را به دست بیاوریم، به شکل زیر است، که با چک کردن همه مسیرها هم برای من تایید شد.

```
NFA
q0
q1 q2 q4
0 1
q0 q1 q2 q3 q4 q5
q3 q1 -
q2 q5 -
q2 q5 -
q0 q4 -
q2 q5 -
q5 q5 -
RegEx
(((1|10(0)*)|0(00)*(01|010(0)*))|0(00)*1(ε|0(0)*))
```

شکل ۱۷: عبارت منظم ماشین اولیه

حال اگر دی اف ای مینیمم را محاسبه کنیم داریم:

```
NFA
q0
q1 q2 q4
0 1
q0 q1 q2 q3 q4 q5
q3 q1 -
q2 q5 -
q2 q5 -
q0 q4 -
q2 q5 -
q5 q5 -
DFA
q0
q1
0 1
q0 q1 q2
q0 q1
q1 q2
q2 q2
```

شکل ۱۸: دی اف ای مینیمم ماشین اولیه

حال اگر عبارت منظم بعد مینیمم شدن را به دست بیاریم :

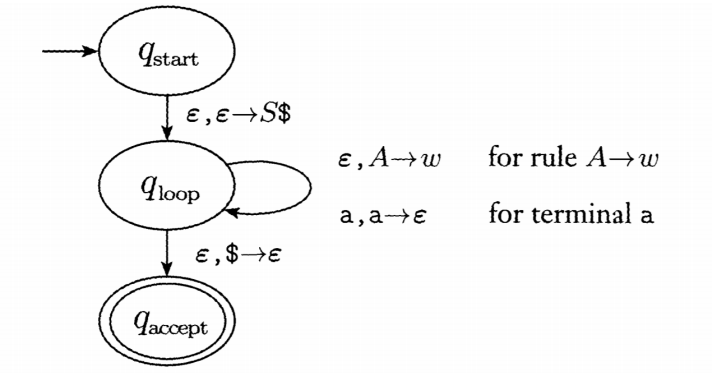
```
NFA
q0
q1
0 1
q0 q1 q2
q0 q1
q1 q2
q2 q2
RegEx
(0)*1(0)*
```

شکل ۱۹: عبارت منظم ماشین مینیمم

منطقا دو عبارت منظمم خروجی شکلهای ۱۹ و ۱۷ یکی هستند؛ بنابراین این جعبه ابزار توانایی ساده سازی عبارات منظم را نیز دارد که البته در این پروژه خواسته نشده بود.

۳ تبدیل گرامر به PDA

برای این کار از الگوریتم کتاب استفاده می‌کنیم. سه حالت در نظر میگیریم؛ qStart، q، qAccept که ارتباط این حالت‌ها در شکل زیر آمده است.



شکل ۲۰: عکس برگرفته از کتاب، روابط بین حالت‌های ساخته شده از CFG

در ابتدا با پشته خالی شروع میکنیم، و سپس با حرکت از qStart به q ابتدا $ و سپس متغیر شروع را در پشته به صورت $\epsilon, \epsilon \to S\$$ وارد می‌کنیم. سپس در حالت q برای هر قاعده به صورت $A \to w$ از q به خودش با $\epsilon, A \to w$ میرویم. برای هر ترمینال مانند a نیز با $a, a \to \epsilon$ از q به خودش می‌رویم. در نهایت نیز با دیدن $ در بالای پشته و بدون خواندن چیزی از ورودی و نوشتن چیزی در پشته به حالت قبول می‌رویم. الفبا Σ همان ترمینالها هستند و الفبا پشته Γ نیز اجتماع متغیرها و ترمینال هاست. ($ را طبق قرارداد جز الفبا پشته لحاظ نمیکنیم.)

۱.۳ توضیح کد

ورودی و خروجی گرفتن با فرمت آمده در توضیحات پروژه است. یک کلاس CFG تعریف شده و با خواندن ورودی اعضای آن مقداردهی می‌شوند و با استفاده از آنها و قواعد گفته شده PDA را می‌سازیم. اعضای کلاس CFG در عکس آمده‌اند :

```
string startVar; // start variable
vector<string> variables; // variables
vector<string> terminals; // terminals
vector<string>* grammers; // grammers, for each variable we have a vector of destinations
int nV, nT; // nV : number of variables , nT : number of terminals
```

شکل ۲۱: متغیرهای CFG

تابع getInput ورودی را میخواند و به متغیرهای بالا مقدار میدهد؛ و تابع toPDA به صورت کاملا واضح PDA مطلوب را می‌سازد. (هر مرحله در کد کامنت گذاری شده است.)

۲.۳ بررسی چند مثال

۱.۲.۳ مثال ۱

خروجی به ازای همان مثال صورت پروژه آورده شده است؛ که مقبول است. (با توجه به اینکه در صورت پروژه قیدی برای وارد کردن چندتایی متغییر در پشته بیان نشده است ما نیز آن را مجاز گرفتیم؛ طبق مباحث بالا نیز خط خالی مربوط به خالی بودن پشته در ابتدا است.)

```
CFG
S
S A
a b
S:aSb|A
A:bAa|S|ε
PDA
qStart

qAccpet
a b
S A a b
qStart q qAccept
9
qStart,ε,ε,q,S$
q,ε,S,q,aSb
q,ε,S,q,A
q,ε,A,q,bAa
q,ε,A,q,S
q,ε,A,q,ε
q,a,a,q,ε
q,b,b,q,ε
q,ε,$,qAccept,ε
```

شکل ۲۲: مثال ۱

۲.۲.۳ مثال ۲

ورودی و خروجی را ملاحظه میکنید که صحیح است. (این مثال مربوط به کتاب است.)

```
CFG
S
S T
a b
S:aTb|b
T:Ta|ε
PDA
qStart

qAccpet
a b
S T a b
qStart q qAccept
8
qStart,ε,ε,q,S$
q,ε,S,q,aTb
q,ε,S,q,b
q,ε,T,q,Ta
q,ε,T,q,ε
q,a,a,q,ε
q,b,b,q,ε
q,ε,$,qAccept,ε
```

شکل ۲۳: مثال ۱