Commentz-Walter

Έχουμε δει αλγορίθμους αναζήτησης συμβολοσειρών οι οποίοι μπορούν γρήγορα να μας βρουν όλες τις εμφανίσεις ενός όρου αναζήτησης μέσα σε ένα κείμενο. Τι γίνεται όμως αν έχουμε παραπάνω από έναν όρο αναζήτησης και θέλουμε να βρούμε όλες τις εμφανίσεις όλων των όρων σε ένα κείμενο; Βεβαίως θα μπορούσαμε να χρησιμοποιήσουμε έναν αλγόριθμο αναζήτησης μιας συμβολοσειράς σε ένα κείμενο και να τον καλέσουμε μία φορά για κάθε συμβολοσειρά που θέλουμε. Μήπως όμως υπάρχει και κάποιος πιο αποτελεσματικός τρόπος ώστε να μπορούμε να αζητήσουμε όλους τους όρους αναζήτησης ταυτόχρονα καθώς σαρώνουμε το κείμενό μας;

Θα εργαστούμε με έναν τρόπο που θυμίζει τον αλγόριθμο Boyer-Moore-Horspool. Έστω ότι έχουμε τους παρακάτω όρους αναζήτησης:

```
cacbaa
acb
aba
acbab
ccbab
```

και ότι θέλουμε να τους αναζητήσουμε στο κείμενο:

```
acbabacacbaa
```

Αν θυμάστε, στον αλγόριθμο Boyer-Moore-Horspool διασχίζουμε τον όρο αναζήτησης κάθε φορά από το τέλος προς την αρχή. Είναι πρακτικό λοιπόν να αντιστρέψουμε τους όρους αναζήτησης:

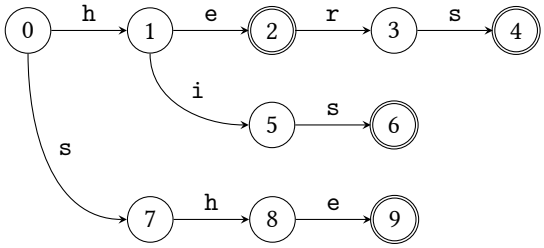
```
cacbaa -> aabcac
acb -> bca
aba -> aba
acbab -> babca
ccbab -> babcc
```

Τώρα θέλουμε έναν τρόπο να αποθηκεύσουμε αυτές τις συμβολοσειρές. Θα θέλαμε η δομή δεδομένων που θα χρησιμοποιήσουμε να μας επιτρέπει να επεξεργαζόμαστε ταυτόχρονα όσες συμβολοσειρές ταιριάζουν στο κείμενο μας, δηλαδή όσες συμβολοσειρές είναι ίδιες, μέχρι το σημείο που εξετάζουμε. Μια τέτοια δομή δεδομένων είναι το trie, ή δένδρο προθεμάτων (prefix tree). Ένα trie είναι ένα δένδρο όπου το μονοπάτι από τη ρίζα σε κάθε φύλλο αναπαριστά μία συμβολοσειρά. Αν έχουμε τις συμβολοσειρές:

```
he
hers
```

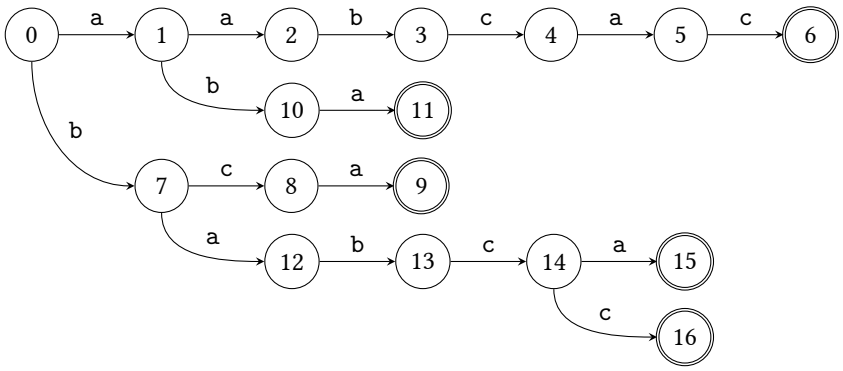
```
his
she
```

το trie που τις αναπαριστά είναι το:



Αν ένας κόμβος είναι κατάληξη μονοπατιού από τη ρίζα που αντιστοιχεί σε μία από τις συμβολοσειρές που αποθηκεύουμε στο trie, τον ονομάζουμε τερματικό κόμβο, και στην εικόνα εμφανίζεται με διπλό περίγραμμα.

Αφού αντιστρέψαμε τους όρους αναζήτησης, μπορούμε να τους εντάξουμε σε ένα trie, το οποίο θα περιδιαβαίνουμε προσπαθώντας να βρούμε ταιριάσματα μέσα στο κείμενο:

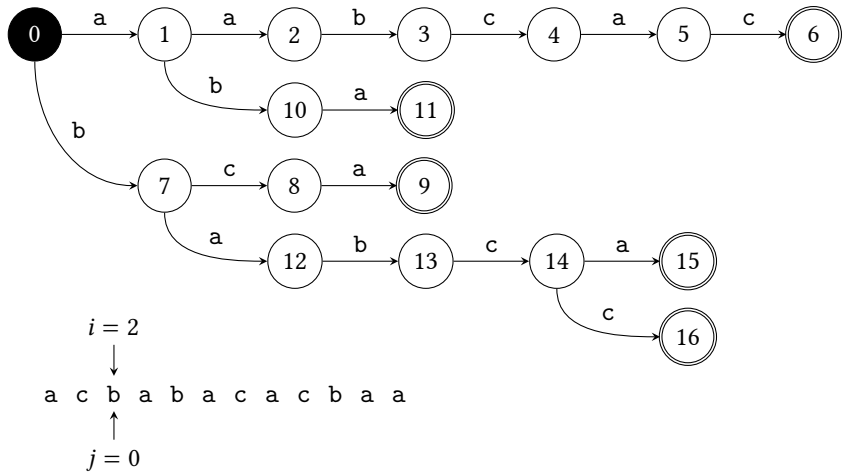


Θέλουμε να βρούμε τους όρους αναζήτησης στο κείμενο:

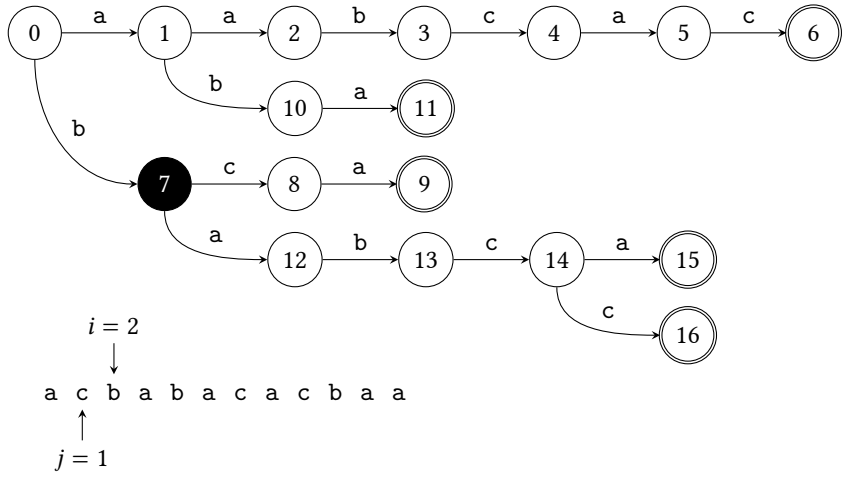
```
acbabacacbaa
```

σαρώνοντας τους όρους αναζήτησης από τα δεξιά προς τα αριστερά. Τότε θα πρέπει να ξεκινήσουμε την προσπάθειά μας *pmin* χαρακτήρες από την αρχή του κειμένου, όπου *pmin* είναι το μήκος του μικρότερου όρου αναζήτησης. Στο παράδειγμά μας, $pmin = 3$, οπότε θα προσπαθήσουμε να δούμε αν στους τρεις πρώτους χαρακτήρες του κειμένου ταιριάζει κάποιος από τους όρους αναζήτησης. Στη συνέχεια θα αναπαριστούμε με i το σημείο που βρισκόμαστε στο κείμενο και με j το βάθος του κόμβου στον οποίο βρισκόμαστε καθώς διασχίζουμε το trie.

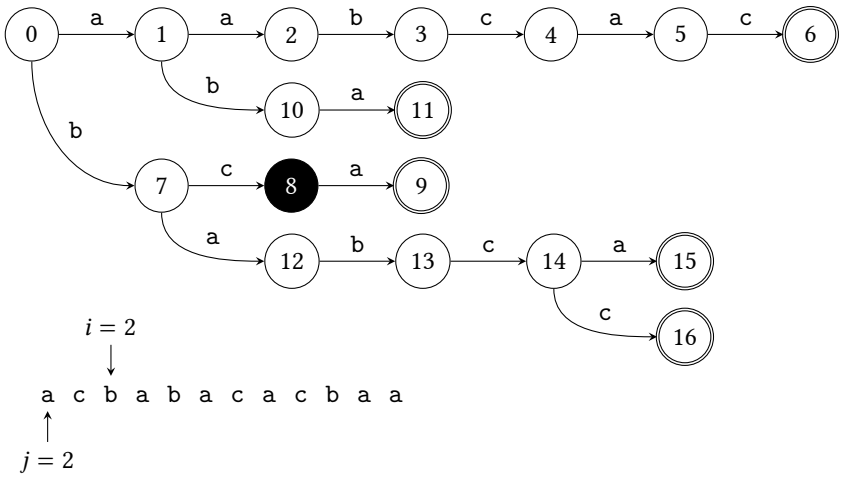
Ξεκινάμε λοιπόν από τη ρίζα του trie. Θέτουμε $i = 2$ και $j = 0$, το οποίο σημαίνει ότι προσπαθούμε να βρούμε ταίριασμα με τους πρώτους τρεις χαρακτήρες του κειμένου ξεκινώντας από τη ρίζα του trie.



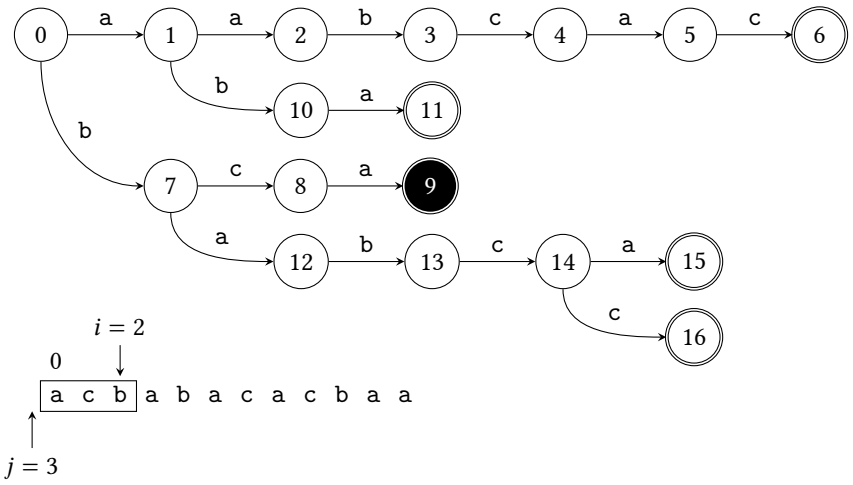
Βλέπουμε ότι μπορούμε να ταιριάξουμε τον χαρακτήρα b προχωρόντας στο trie από τον κόμβο 0 στον κόμβο 7, και έχοντας πλέον $j = 1$.



Τώρα μπορούμε να ταιριάξουμε τον χαρακτήρα c προχωρόντας στο trie από τον κόμβο 7 στον κόμβο 8, και έχοντας $j = 2$.

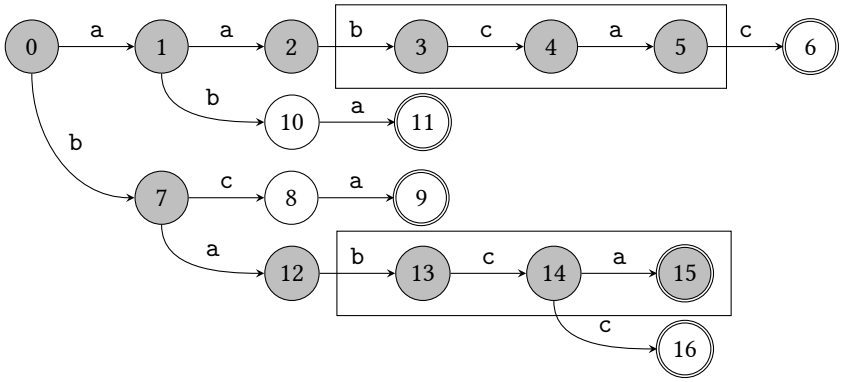


Στο σημείο αυτό μπορούμε να ταιριάξουμε τον χαρακτήρα a προχωρόντας στο trie από τον κόμβο 8 στον κόμβο 9. Ο κόμβος 9 είναι τερματικός κόμβος στο trie, άρα ταιριάξαμε έναν όρο αναζήτησης. Ο όρος αναζήτησης που ταιριάξαμε είναι αυτός που προκύπτει από το αντεστραμμένο μονοπάτι από τη ρίζα μέχρι τον τερματικό κόμβο που βρισκόμαστε, και το ταίριασμα έγινε στη θέση $i - j + 1 = 0$ του κειμένου.

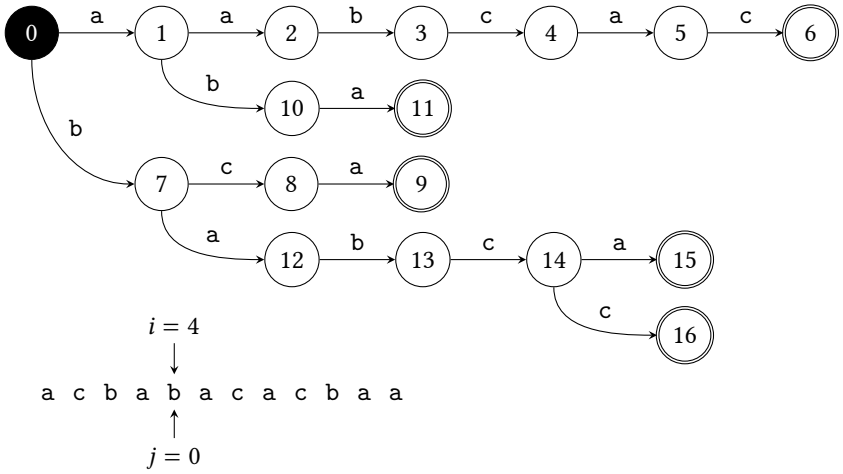


Έχοντας βρει το acb στην αρχή του κειμένου μπορούμε να συνεχίσουμε την αναζήτηση. Αφού έχουμε περισσότερους από έναν όρους αναζήτησης, έχοντας βρει το acb, είναι πιθανό να μπορούμε να βρούμε έναν άλλο όρο αναζήτησης, στο ίδιο σημείο του κειμένου, ο οποίος να ξεκινάει με acb. Ο όρος αυτός θα έχει το acb ως πρόθεμα. Τότε όμως ο *ανίστροφος* του όρου αυτού θα έχει το bca ως κατάληξη. Επομένως, θα πρέπει να δούμε αν τυχόν στο trie υπάρχουν άλλα μονοπάτια, εκτός αυτού που βρήκαμε ήδη, που να καταλήγουν σε bca. Πράγματι, υπάρχουν δύο τέτοια μονοπάτια, το bca είναι κατάληξη του μονοπατιού $0 \to 1 \to 2 \to 3 \to 4 \to 5$ (aabca) και του

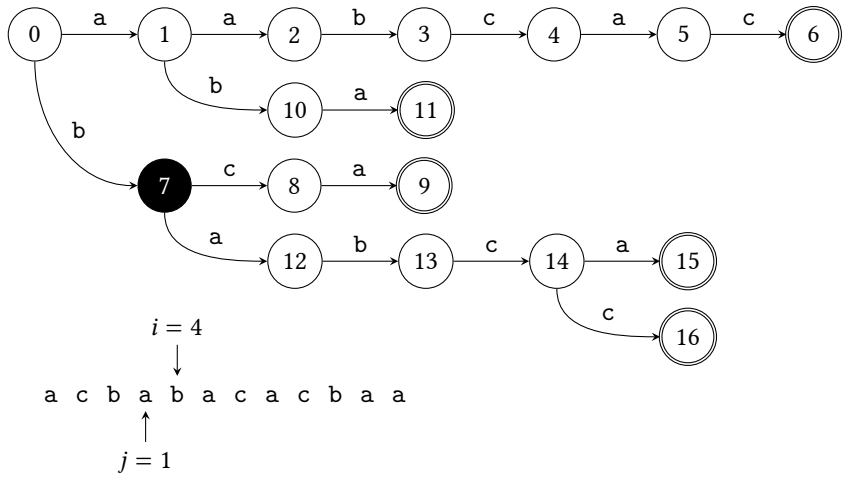
μονοπατιού $0 \to 7 \to 12 \to 13 \to 14 \to 15$ (babca).



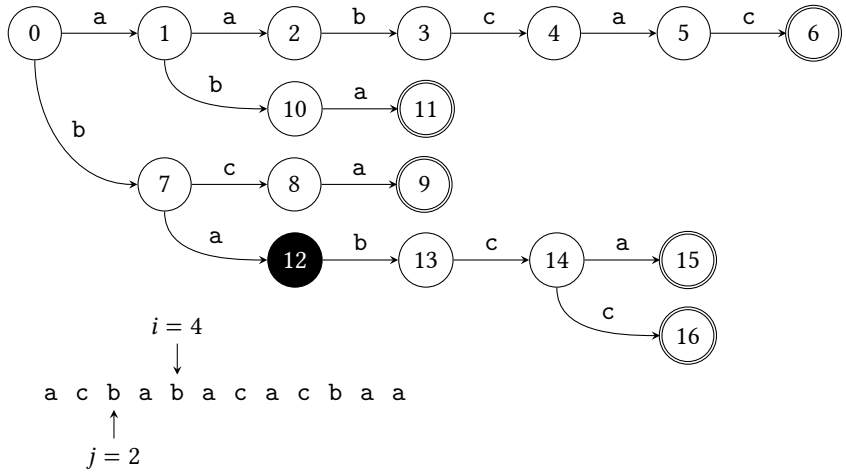
Άρα θέλουμε να δούμε αν τυχόν μπορούμε να ταιριάξουμε το acbaa ή το acbab στο κείμενό μας. Για να κάνουμε αυτό θα πρέπει να κάνουμε μια ολίσθηση δύο χαρακτήρων, και ξεκινάμε μια προσπάθεια ταιριάσματος, θέτοντας $i \leftarrow i + 2$ και $j \leftarrow 0$. Ελπίζουμε δηλαδή με την ολίσθηση των δύο χαρακτήρων να μπορέσουμε να ταιριάξουμε τους δύο επόμενους χαρακτήρες στο κείμενο και στη συνέχεια να πέσουμε στους acb που ξέρουμε ότι μπορούμε να ταιριάξουμε στα δύο παραπάνω μονοπάτια.



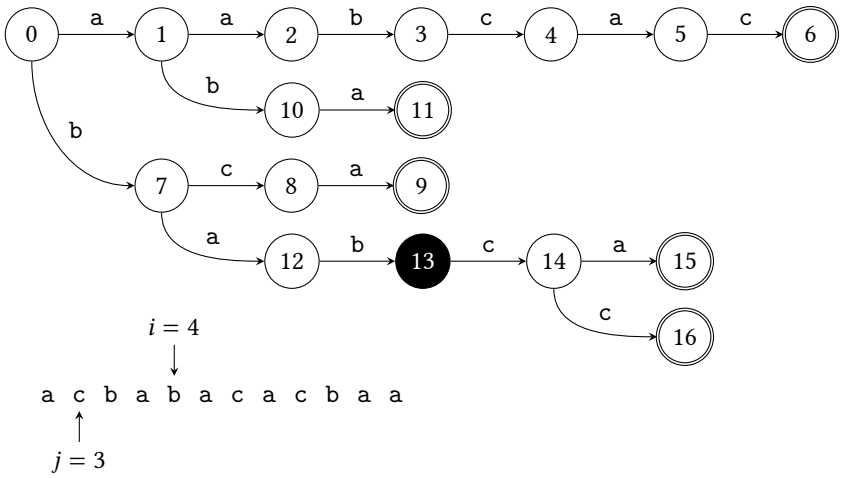
Βλέπουμε ότι μπορούμε να ταιριάξουμε τον χαρακτήρα b προχωρόντας στο trie από τον κόμβο 0 στον κόμβο 7, και έχοντας πλέον $j = 1$.



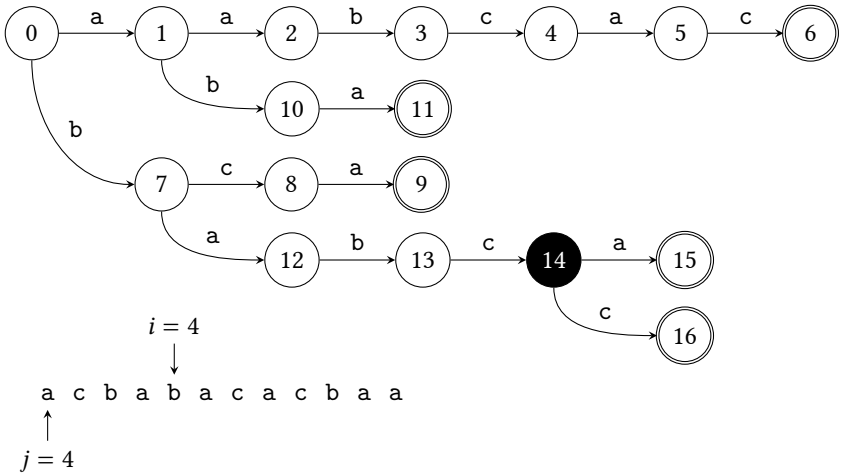
Συνεχίζουμε στον κόμβο 12 ταιριάζοντας τον χαρακτήρα a.



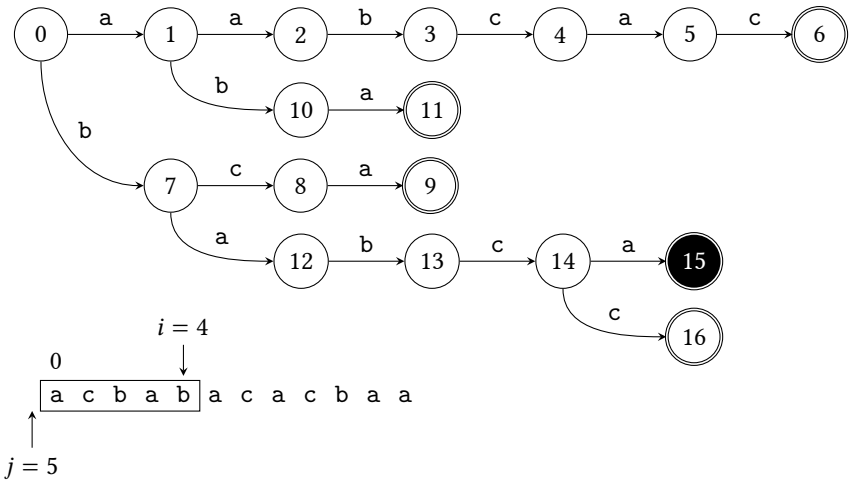
Προχωράμε στον κόμβο 13 ταιριάζοντας τον χαρακτήρα b.



Κατόπιν στον κόμβο 14 ταιριάζοντας τον χαρακτήρα c.

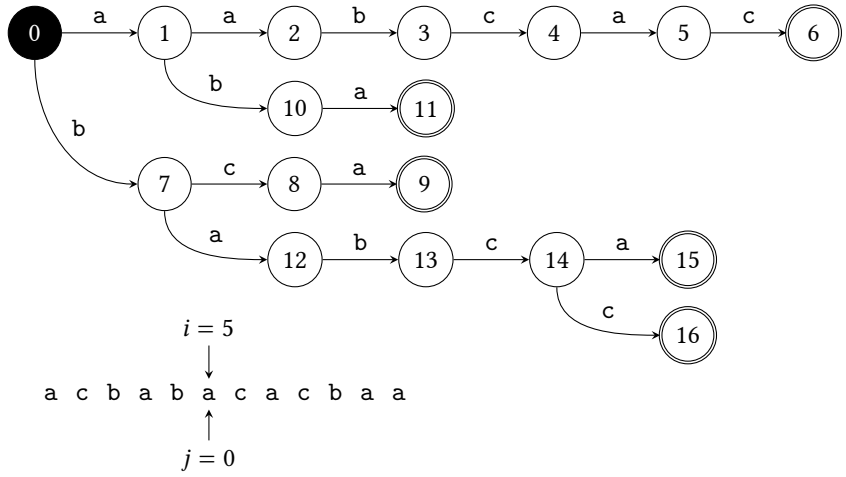


Εδώ μπορούμε να ταιριάξουμε τον χαρακτήρα a προχωρόντας στο trie από τον κόμβο 14 στον κόμβο 15 και διαπιστώνουμε ότι βρήκαμε ταίριασμα για τον όρο αναζήτησης acbab στη θέση $i - j + 1 = 0$ του κειμένου.

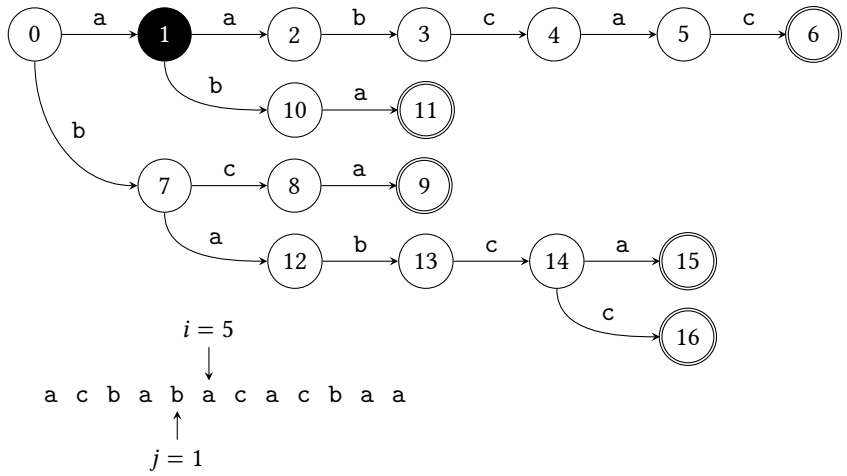


Αφού ταιριάξαμε στο acbab πρέπει να δούμε πού θα μεταφερθούμε στο κείμενο, δηλαδή πόσο θα προσαρμόσουμε το i. Σκεφτόμενοι όπως προηγουμένως, βλέπουμε ότι στο trie δεν υπάρχει κάποιο βαθύτερο μονοπάτι που να περιέχει το babca. Πλην όμως με ολίσθηση ενός χαρακτήρα μπορούμε να δοκιμάσουμε να ταιριάξουμε το aba που αντιστοιχεί στο μονοπάτι $0 \to 1 \to 10 \to 11$, δεδομένου ότι το aba περιέχει το ba.

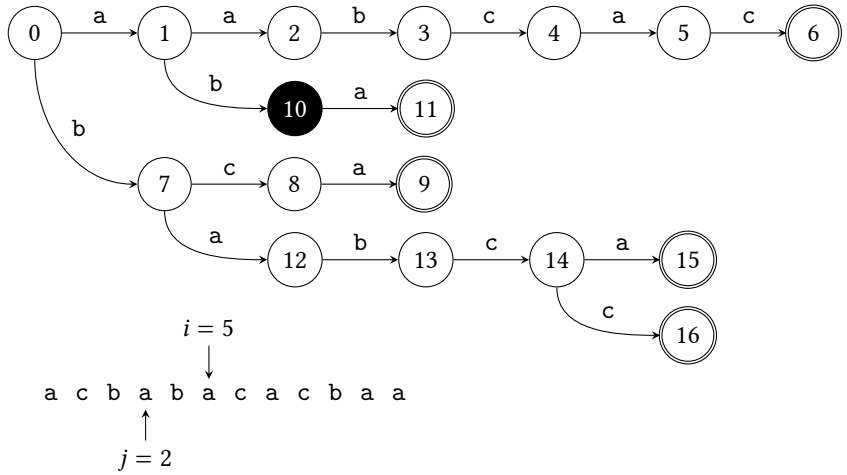
Αν σκεφτούμε τι θέλουμε να πετύχουμε, η λογική είναι ότι προσπαθούμε να χρησιμοποιήσουμε για ταίριασμα τη μεγαλύτερη δυνατή κατάληξη του acbab. Αυτή που μπορούμε να χρησιμοποιήσουμε είναι η ab, αντεστραμμένη είναι ba, και αυτή περιέχεται στο aba. Ας προχωρήσουμε λοιπόν με ολίσθηση ενός χαρακτήρα.



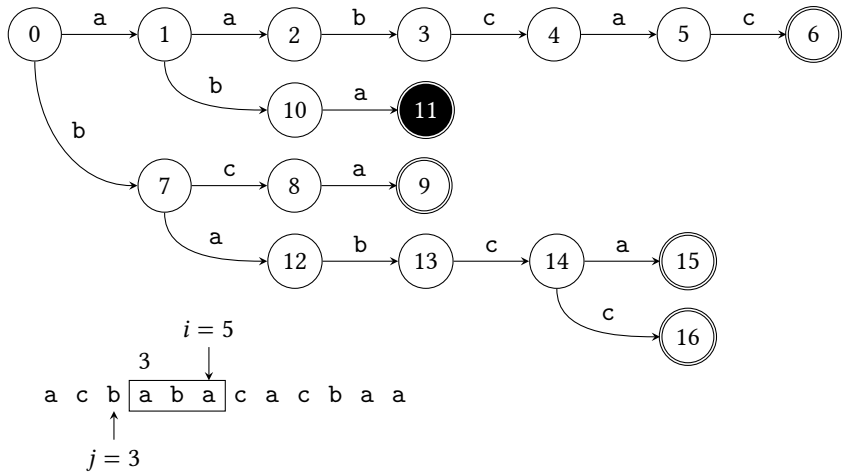
Πράγματι, μπορούμε να ταιριάξουμε τον χαρακτήρα a πηγαίνοντας από τη ρίζα του trie στον κόμβο 1.



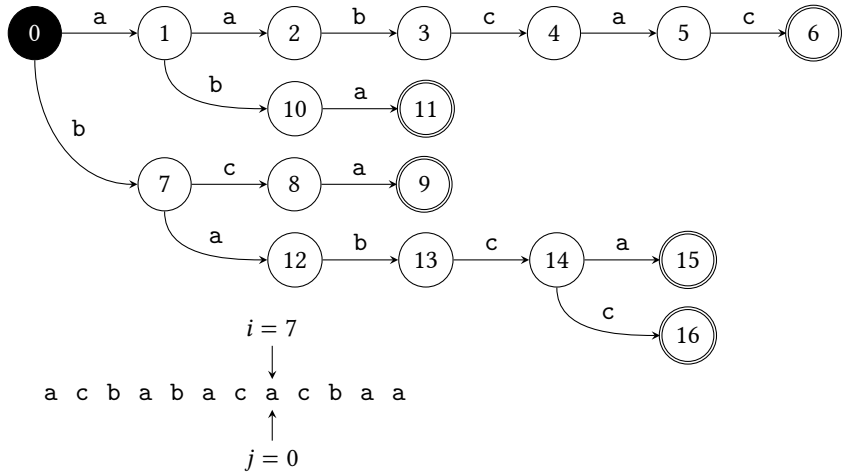
Συνεχίζουμε ταιριάζοντας το γράμμα b πηγαίνοντας από τον κόμβο 1 στον κόμβο 10.



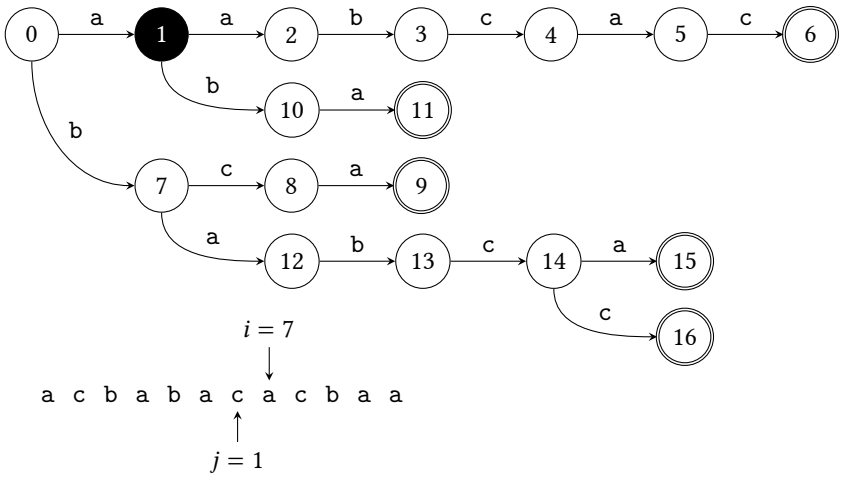
Μπορούμε να ταιριάξουμε τον χαρακτήρα a προχωρόντας στο trie από τον κόμβο 14 στον κόμβο 15 και διαπιστώνουμε ότι βρήκαμε ταίριασμα για τον όρο αναζήτησης aba στη θέση $i - j + 1 = 3$ του κειμένου.



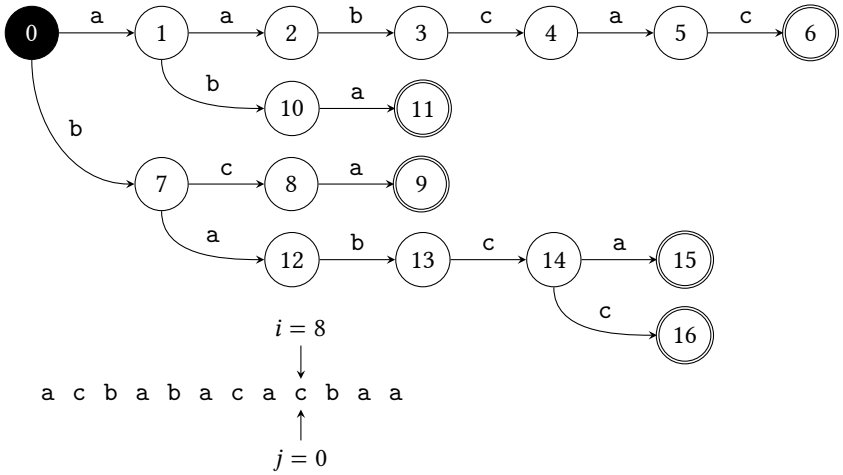
Έχοντας ταιριάξει το aba, παρατηρούμε ότι δεν υπάρχει άλλο βαθύτερο μονοπάτι στο trie το οποίο να περιέχει το aba. Αν κάναμε μια ολίσθηση ενός χαρακτήρα, θα προσπαθούσαμε να βρούμε ένα έναν όρο αναζήτησης με τρεις χαρακτήρες που να αρχίζει με ba, το οποίο αντιστοιχεί σε ένα μονοπάτι τριών χαρακτήρων στο trie που να έχει κατάληξη ab. Κάτι τέτοιο δεν υπάρχει. Μπορούμε να δοκιμάσουμε με μία ολίσθηση δύο χαρακτήρων, δηλαδή να προσπαθήσουμε να ταιριάξουμε έναν όρο αναζήτησης με τρεις χαρακτήρες που να τελειώνει σε a. Τέτοιος όρος υπάρχει, είναι πάλι ο aba, οπότε ας δοκιμάσουμε με ολίσθηση δύο χαρακτήρων.



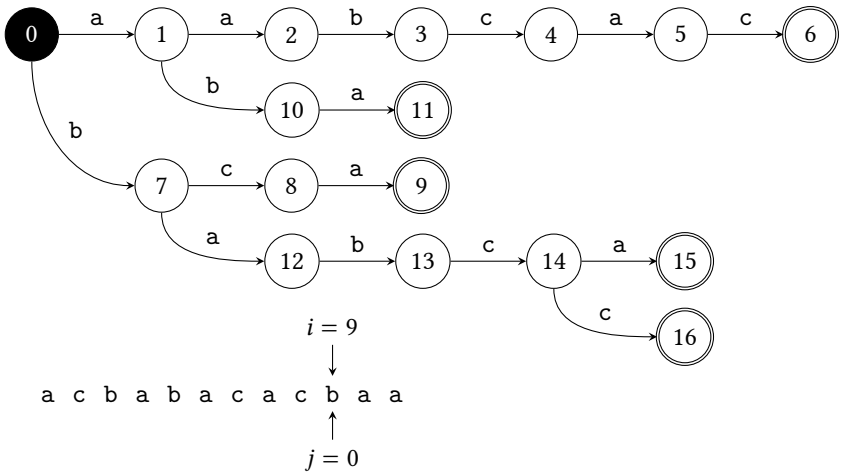
Ξεκινάμε τη διάσχιση του trie ταιριάζοντας τον χαρακτήρα a και πηγαίνοντας στον κόμβο 1.



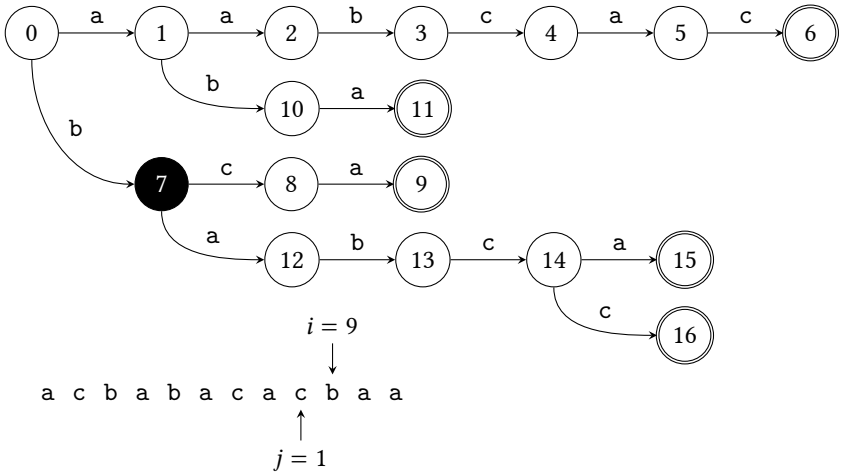
Εδώ όμως βλέπουμε ότι δεν μπορούμε να ταιριάξουμε τον χαρακτήρα c. Έχοντας ταιριάξει μόνο τον χαρακτήρα a, μπορούμε να δοκιμάσουμε μονοπάτια στο trie με τον χαρακτήρα a στη δεύτερη θέση, που αντιστοιχεί σε ολίσθηση ενός χαρακτήρα στο κείμενό μας.



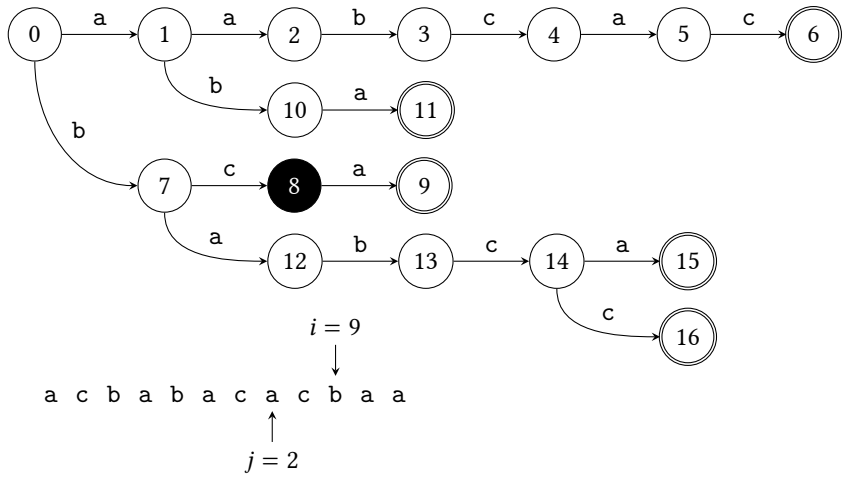
Δυστυχώς όμως δεν υπάρχει κανένα μονοπάτι το οποίο να ξεκινάει από τον χαρακτήρα c, το c όμως εμφανίζεται ως δεύτερος χαρακτήρας στο μονοπάτι $0 \to 7 \to 8$, άρα μπορούμε να ολισθήσουμε πάλι κατά έναν χαρακτήρα.



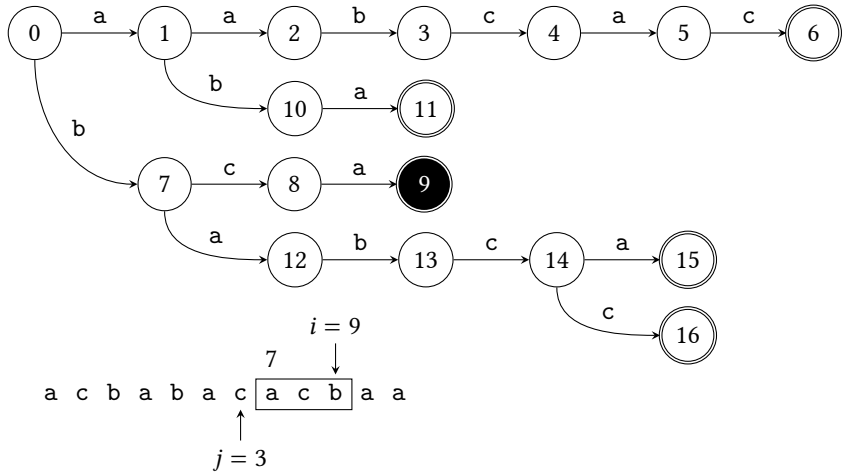
Προχωράμε στον κόμβο 7 ταιριάζοντας τον χαρακτήρα b.



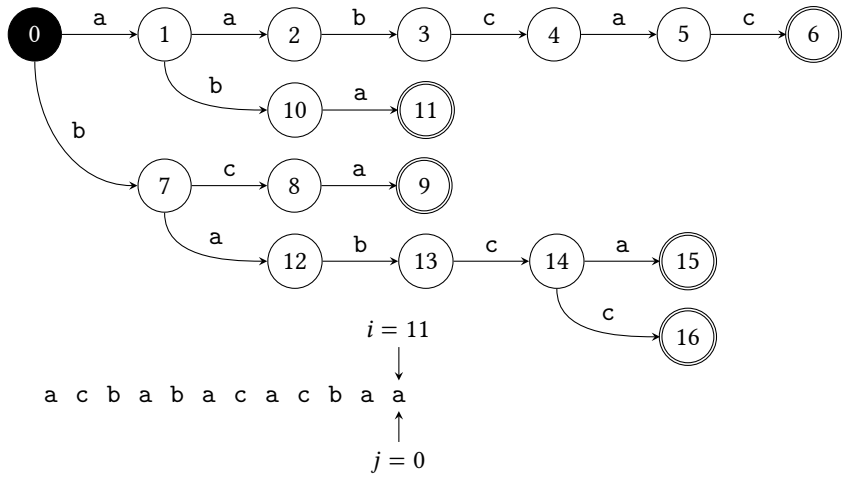
Από τον κόμβο 7 πηγαίνουμε στον κόμβο 8 ταιριάζοντας τον χαρακτήρα c.



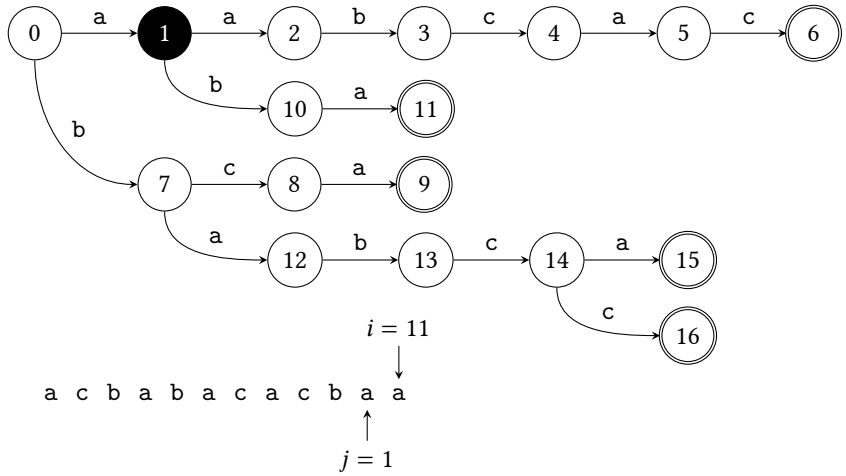
Από τον κόμβο 8 πηγαίνουμε στον κόμβο 9 ταιριάζοντας τον χαρακτήρα a. Διαπιστώνουμε ότι βρήκαμε ταίριασμα για τον όρο αναζήτησης acb στη θέση $i - j + 1 = 7$ του κειμένου.



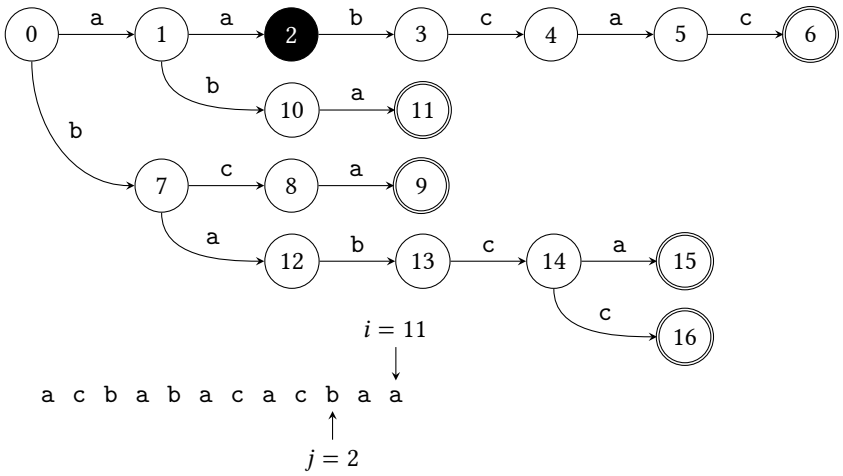
Έχοντας ταιριάξει το acb, μπορούμε να παρατηρήσουμε ότι το αντιστροφό του bca περιέχεται σε άλλα μονοπάτια του trie, συγκεκριμένα το μονοπάτι $0 \to 1 \to 2 \to 3 \to 4 \to 5$ (aabca) και $0 \to 7 \to 12 \to 13 \to 14$ (babca). Και τα δύο αυτά βρίσκονται σε βάθος δύο στο trie, άρα μπορούμε να συνεχίσουμε με μία ολίσθηση δύο χαρακτήρων.



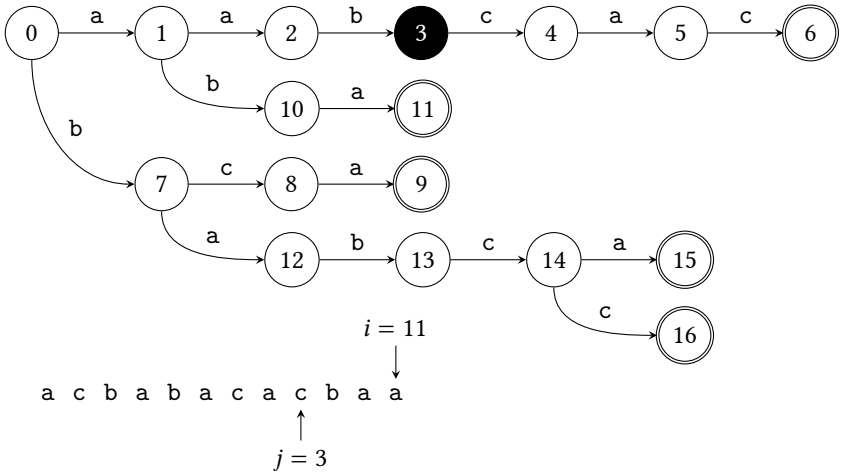
Προχωράμε στον κόμβο 1 ταιριάζοντας τον χαρακτήρα a.



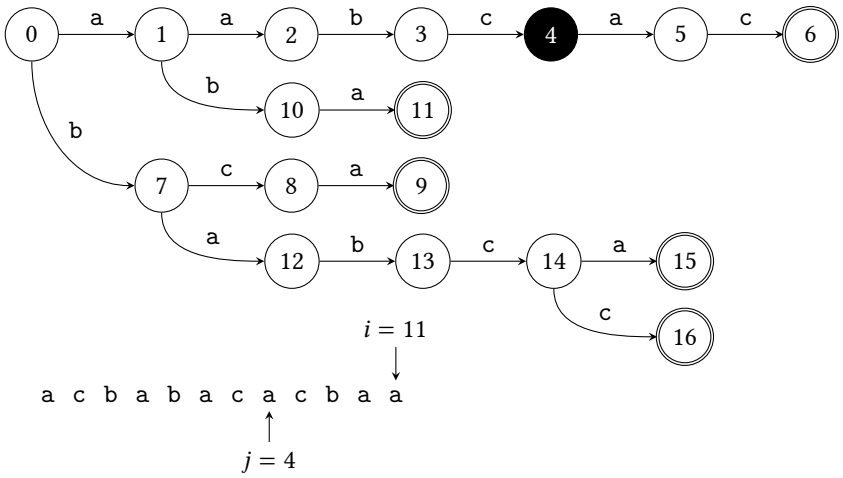
Συνεχίζουμε στον κόμβο 2 ταιριάζοντας τον χαρακτήρα a.



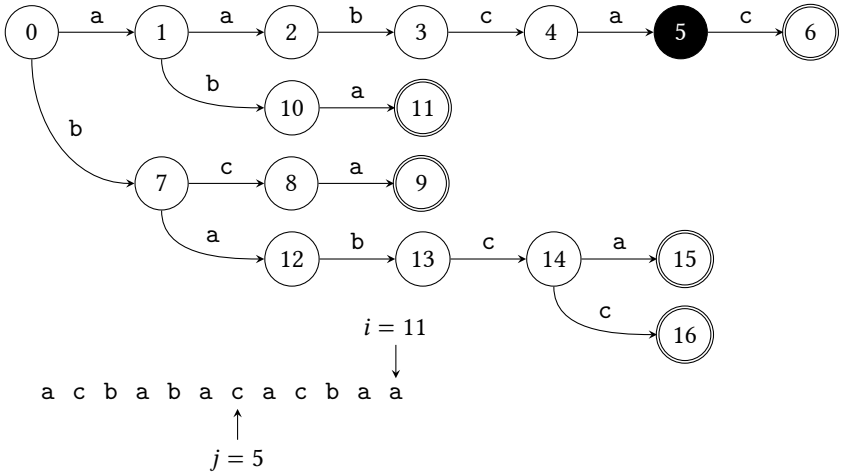
Από τον κόμβο 2 προχωράμε στον κόμβο 3 ταιριάζοντας τον χαρακτήρα b.



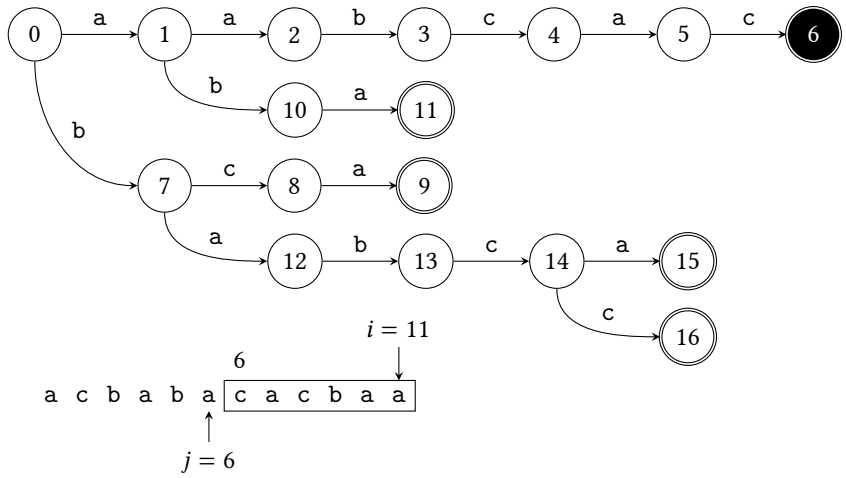
Στη συνέχεια από τον κόμβο 3 προχωράμε στον κόμβο 4 ταιριάζοντας τον χαρακτήρα c.



Περνώντας στον κόμβο 5 ταιριάζουμε τον χαρακτήρα a.



Έτσι φτάνουμε στον κόμβο 6 ταιριάζοντας τον χαρακτήρα c. Διαπιστώνουμε τότε ότι βρήκαμε ταίριασμα για τον όρο αναζήτησης cacbaa στη θέση $i - j + 1 = 6$ του κειμένου. Έχουμε φτάσει στο τέλος του κειμένου, έχοντας βρει όλες τις εμφανίσεις όλων των όρων αναζήτησης.



Η διαδικασία που περιγράψαμε ακολουθεί τον αλγόριθμο Commentz-Walter που μπορείτε να δείτε παρακάτω, και τον οποίο θα πρέπει να υλοποιήσετε στην εργασία.

Ο αλγόριθμος ξεκινάει δημιουργώντας μια ουρά όπου θα αποθηκεύσουμε τα αποτελέσματα, δηλαδή για κάθε ταίριασμα, τον όρο αναζήτησης και το σημείο που βρέθηκε. Στη συνέχεια ορίζουμε τη μεταβλητή i, η οποία δείχνει τη θέση που βρισκόμαστε στο κείμενο, την μεταβλητή j, η οποία δείχνει το βάθος στο οποίο βρισκόμαστε στο trie, τη μεταβλητή u, η οποία δείχνει τον κόμβο που βρισκόμαστε στο trie, και τη μεταβλητή m που περιέχει το τρέχον ταίριασμα.

Το κύριο μέρος του αλγορίθμου, γραμμές 6–19, εκτελείται όσο δεν έχουμε φτάσει στο τέλος του κειμένου μας. Προχωράμε χαρακτήρα-χαρακτήρα και από τα δεξιά προς τα αριστερά, όσο αυτός ταιριάζει με το σημείο που βρισκόμαστε στο trie (γραμμές 7–12). Η κλήση `HasChild(`*trie*, *u*, $t[i-j]$`)` αποφαίνεται αν ο κόμβος u του *trie* έχει παιδί μέσω κλαδιού επισημειωμένου με τον τρέχοντα χαρακτήρα $t[i-j]$ του κειμένου. Αν ναι, η κλήση `GetChild(`*trie*, *u*, $t[i-j]$`)` μας επιστρέφει το εν λόγω παιδί. Προσθέτουμε τον χαρακτήρα στο τρέχον ταίριασμα, και αν φτάσουμε σε τερματικό κόμβο προσθέτουμε στην ουρά το ταίριασμα και τη θέση που το βρήκαμε.

Όταν δεν μπορούμε να ταιριάξουμε άλλο χαρακτήρα ακολουθώντας το trie, είτε επειδή φτάσαμε στην αρχή του trie είτε επειδή προέκυψε διαφωνία, πρέπει να δούμε πόσο θα ολισθήσουμε. Στο μεταξύ, αν έχουμε φτάσει στην αρχή του trie φροντίζουμε ώστε η μεταβλητή j να μην δείχνει πριν από την αρχή (γραμμές 13–14). Η ολίσθηση προκύπτει από την έκφραση:

$$s = \min(s_2[u], \max(s_1[u], rt[t[i-j]] - j - 1))$$

Τα περιέχομενα της έκφρασης αυτή θα εξηγήσουμε στη συνέχεια. Προσαρμόζουμε ανάλογα τις μεταβλητές i, j, u, και m και συνεχίζουμε μέχρι εξάντλησης του κειμένου μας.

`CommentzWalter(`t`)` $\rightarrow q$

Input: t, το κείμενο στο οποίο θα αναζητηθούν οι όροι αναζήτησης

Data: *trie*, ένα trie που έχει κατασκευαστεί από τους αντεστραμμένους όρους αναζήτησης
pmin, το μικρότερο μήκος όρου αναζήτησης
rt, ο πίνακας των δεξιότερων εμφανίσεων των χαρακτήρων των όρων αναζήτησης
s_1, ο πίνακας s_1 που έχει κατασκευαστεί όπως έχουμε ορίσει
s_2, ο πίνακας s_2 που έχει κατασκευαστεί όπως έχουμε ορίσει

Output: q, ουρά με ζευγάρια (m, i), όπου m όρος αναζήτησης και i η θέση που βρέθηκε στο t.

```
1  q ← CreateQueue()
2  i ← pmin − 1
3  j ← 0
4  u ← 0
5  m ← ''
6  while i < |t| do
7      while HasChild(trie, u, t[i − j]) do
8          u ← GetChild(trie, u, t[i − j])
9          m ← m + t[i − j]
10         j ← j + 1
11         if IsTerminal(trie, u) = TRUE then
12             Enqueue(q, (ReverseString(m), i − j + 1))
13     if j > i then
14         j ← i
15     s ← Min(s2[u], Max(s1[u], rt[t[i − j]] − j − 1))
16     i ← i + s
17     j ← 0
18     u ← 0
19     m ← ''
20 return q
```

Ο αλγόριθμος χρησιμοποιεί μια σειρά δεδομένων και δομών για την εκτέλεσή του. Ας τα δούμε αυτά με τη σειρά.

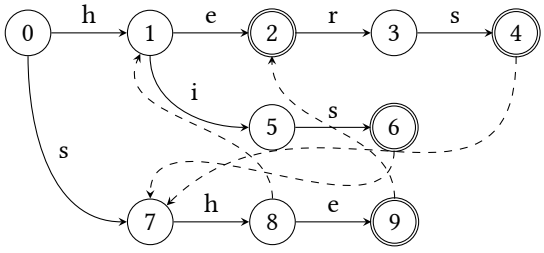
Η μεταβλητή *trie* είναι το trie το οποίο έχουμε κατασκευάσει με τους αντεστραμμένους όρους αναζήτησης. Η μεταβλητή *pmin* περιέχει το μικρότερο μήκος όρου αναζήτησης. Ο πίνακας *rt* είναι ο πίνακας δεξιότερων εμφανίσεων χαρακτήρων που γνωρίζουμε από τον αλγόριθμο Boyer-Moore-Horspool, μόνο που καλύπτει όλους τους χαρακτήρες από όλους τους όρους αναζήτησης όπως τους αποθηκεύουμε στο trie, και προσθέτουμε μια μονάδα στην τιμή του πίνακα για τους χαρακτήρες που δεν υπάρχουν στους όρους αναζήτησης. Με $d(u)$ θα συμβολίζουμε το βάθος του κόμβου u στο trie και με $l(u)$ τον χαρακτήρα που αντιπροσωπεύει ο κόμβος u στο trie. Τότε ο πίνακας *rt* είναι:

$$rt[char] = \min(pmin + 1, \{d(u) \,:\, l(u) = char\})$$

Στο παράδειγμά μας έχουμε:

$$rt[char] = \begin{cases} 1, & char = \texttt{a} \\ 1, & char = \texttt{b} \\ 2, & char = \texttt{c} \\ 4, & \text{για τους υπόλοιπους χαρακτήρες του αλφαβήτου} \end{cases}$$

Για να μπορεί να εκτελεστεί ο αλγόριθμος, θα πρέπει τώρα να ορίσουμε το πώς κατασκευάζονται οι πίνακες s_1 και s_2 στην έκφραση που είδαμε. Για τη δημιουργία των πινάκων αυτών πρέπει πρώτα να φτιάξουμε έναν άλλο βοηθητικό πίνακα ο οποίος ονομάζεται *failure*. Με $w(u)$ θα συμβολίζουμε τη συμβολοσειρά που προκύπτει ακολουθώντας το μονοπάτι από τη ρίζα του trie μέχρι τον κόμβο u. Τότε ο πίνακας *failure* μας δίνει για κάθε κόμβο u, έναν κόμβο $failure[u] = v$, ώστε το $w(v)$ να είναι μια όσο το δυνατόν μεγαλύτερη κατάληξη του $w(u)$, αν υπάρχει. Για την περίπτωση του παρακάτω trie, τα μη μηδενικά περιεχόμενα του πίνακα *failure* εμφανίζονται με διακεκομμένες γραμμές.



Ο πίνακας failure είναι:

i	0	1	2	3	4	5	6	7	8	9
$failure[i]$	0	0	0	0	7	0	7	0	1	2

Για να κατασκευάσουμε τον πίνακα *failure* εργαζόμαστε ως εξής:

1. Θέτουμε $failure[u] \leftarrow 0$ για κάθε κόμβο u σε βάθος $d \leq 1$ του trie (η ρίζα είναι στο επίπεδο μηδέν).

2. Διατρέχουμε το trie με μία κατά πλάτος αναζήτηση. Για κάθε κόμβο u που επισκεπτόμαστε στην κατά πλάτος αναζήτηση σε βάθος $d \geq 1$, εξετάζουμε ένα-ένα τα παιδιά του. Για κάθε παιδί v που συνδέεται με τον u με έναν σύνδεσμο c:

   2.1 Θέτουμε $u' \leftarrow failure[u]$.

   2.2 Όσο ο κόμβος u' στο trie δεν έχει παιδί με σύνδεσμο c, και ο u' δεν είναι η ρίζα, θέτουμε $u' \leftarrow failure[u']$ (δηλαδή ανεβαίνουμε προς τη ρίζα του δένδρου εξετάζοντας τους προγόνους του v).

   2.3 Αν βρήκαμε πρόγονο κόμβο u' με παιδί v' μέσω συνδέσμου c, θέτουμε $failure[v] \leftarrow v'$. Διαφορετικά, θέτουμε $failure[v] \leftarrow 0$.

Στην περίπτωση του trie για τους όρους αναζήτησης `cacbaa`, `acb`, `aba`, `acbab`, `ccbab`, ο πίνακας *failure* είναι:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$failure[i]$	0	0	1	10	8	9	0	0	0	1	7	12	1	10	8	9	0

Παρατηρήστε ότι $failure[11] = 12$ και όχι 1. Αφού κατασκευάσουμε τον πίνακα *failure*, κατασκευάζουμε τον πίνακα set_1 ως εξής:

$$set_1[u] = \{v : failure[v] = u\}$$

δηλαδή ο set_1 είναι η αντιστροφή του *failure*. Για τον προηγούμενο πίνακα *failure* θα έχουμε:

i	1	7	8	9	10	12
$set_1[i]$	$\{2, 9, 12\}$	$\{10\}$	$\{4, 14\}$	$\{5, 15\}$	$\{3, 13\}$	$\{11\}$

Το $set_1[u]$ περιέχει τους πιο κοντινούς κόμβους του u που είναι σε βαθύτερο επίπεδο του trie από τον u και $w(u)$ είναι η μεγαλύτερη δυνατή κατάληξη του $w(v)$, για κάθε $v \in set_1[u]$. Για το τελευταίο σημείο, παρατηρείστε ότι $11 \in set_1[12]$ ενώ $11 \notin set_1[1]$.

Από τον πίνακα set_1 κατασκευάζουμε μετά τον πίνακα set_2:

$$set_2[u] = \{v : v \in set_1[u] \text{ και ο } v \text{ είναι τερματικός κόμβος στο trie}\}$$

το $set_2[u]$ είναι το υποσύνολο του $set_1[u]$ για το οποίο κάθε μονοπάτι των κόμβων του $set_1[u]$ είναι ένας όρος αναζήτησης. Για το προηγούμενο set_1 το set_2 είναι:

i	1	9	12
$set_2[i]$	$\{9\}$	$\{15\}$	$\{11\}$

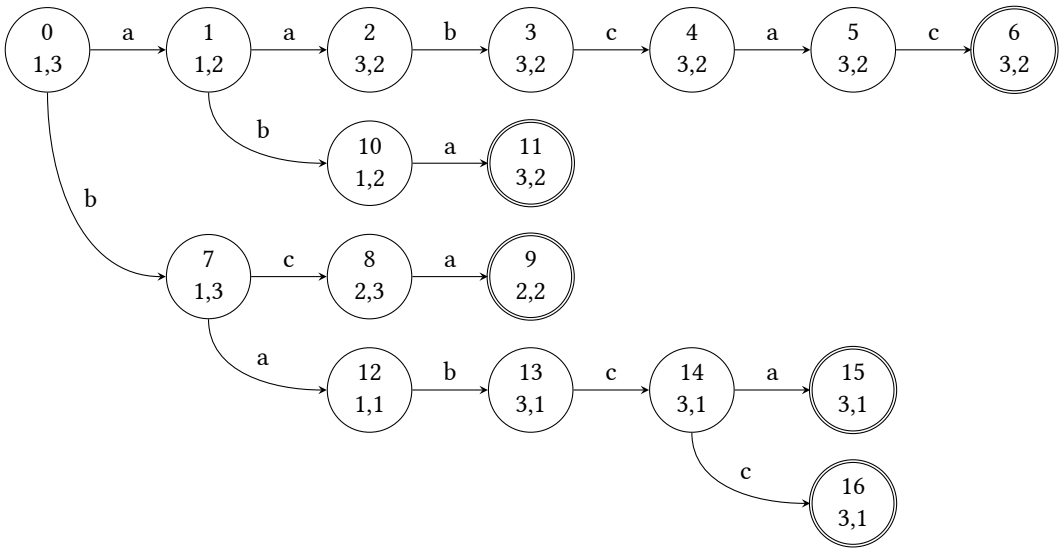
Από τους set_1 και set_2 τώρα μπορούμε να κατασκευάσουμε τους s_1 και s_2. Στους παρακάτω ορισμούς, r είναι η ρίζα του trie και $parent(u)$ είναι ο γονέας του u στο trie. Η έκφραση $k : k = d(u') - d(u), u' \in set_1[u]$ μας δίνει για κάθε κόμβο u' του $set_1[u]$

τη διαφορά του βάθους του από το βάθος του u, και ομοίως η αντίστοιχη έκφραση στο s_2.

$$s_1[u] = \begin{cases} 1, & u = r \\ \min\{pmin, k \,:\, k = d(u') - d(u), u' \in set_1[u]\} & u \neq r \end{cases}$$

$$s_2[u] = \begin{cases} pmin, & u = r \\ \min\{s_2[parent(u)], k \,:\, k = d(u') - d(u), u' \in set_2[u]\} & u \neq r \end{cases}$$

Στην παρακάτω εικόνα εμφανίζεται το trie του παραδείγματός μας, μαζί με τις τιμές των s_1 και s_2 σε κάθε κόμβο.



Απαιτήσεις Προγράμματος

Κάθε φοιτητής θα εργαστεί σε αποθετήριο στο GitHub. Για να αξιολογηθεί μια εργασία θα πρέπει να πληροί τις παρακάτω προϋποθέσεις:

- Για την υποβολή της εργασίας θα χρησιμοποιηθεί το ιδιωτικό αποθετήριο του φοιτητή που δημιουργήθηκε για τις ανάγκες του μαθήματος και του έχει αποδοθεί. Το αποθετήριο αυτό έχει όνομα του τύπου `username-algo-assignments`, όπου `username` είναι το όνομα του φοιτητή στο GitHub. Για παράδειγμα, το σχετικό αποθετήριο του διδάσκοντα θα ονομαζόταν `louridas-algo-assignments` και θα ήταν προσβάσιμο στο https://github.com/dmst-algorithms-course/louridas-algo-assignments. *Τυχόν άλλα αποθετήρια απλώς θα αγνοηθούν.*
- Μέσα στο αποθετήριο αυτό θα πρέπει να δημιουργηθεί ένας κατάλογος `assignment-2023-3`.

- Μέσα στον παραπάνω κατάλογο το πρόγραμμα θα πρέπει να αποθηκευτεί με το όνομα `commentz_walter.py`.
- Δεν επιτρέπεται η χρήση έτοιμων βιβλιοθηκών γράφων ή τυχόν έτοιμων υλοποιήσεων των αλγορίθμων, ή τμημάτων αυτών, εκτός αν αναφέρεται ρητά ότι επιτρέπεται.
- Επιτρέπεται η χρήση δομών δεδομένων της Python όπως στοίβες, λεξικά, σύνολα, κ.λπ.
- Επιτρέπεται η χρήση των παρακάτω βιβλιοθηκών ή τμημάτων τους όπως ορίζεται:
  - `sys.argv`
  - `argparse`
  - `deque`
- Το πρόγραμμα θα πρέπει να είναι γραμμένο σε Python 3.
- Το πρόγραμμα θα πρέπει να υλοποιεί τον αλγόριθμο Commentz-Walter όπως περιγράφεται εδώ. Τυχόν άλλες υλοποιήσεις δεν θα γίνουν δεκτές.
- Η έξοδος του προγράμματος θα πρέπει να περιλαμβάνει μόνο ό,τι φαίνεται στα παραδείγματα. *Η φλυαρία δεν επιβραβεύεται.*

Το πρόγραμμα θα καλείται ως εξής (όπου `python` η κατάλληλη εντολή στο εκάστοτε σύστημα):

```
python comments_walter.py [-v] kw [kw ...] input_filename
```

Η σημασία των παραμέτρων είναι η εξής:

- Η παράμετρος `-v`, αν υπάρχει, σημαίνει ότι το πρόγραμμα στην έξοδο θα εμφανίσει, πλέον της κανονικής εξόδου, τα περιεχόμενα των s_1 και s_2 για κάθε κόμβο.
- Το `kw [kw ...]` σημαίνει ότι δίνουμε έναν ή περισσότερους όρους αναζήτησης (keywords, kw).
- Η παράμετρος `input_filename` είναι το όνομα του αρχείου στο οποίο θέλουμε να αναζητήσουμε τους όρους αναζήτησης.

Το πρόγραμμα θα εμφανίζει κάθε όρο αναζήτησης και το σημείο που βρέθηκε αυτός.

Παραδείγματα

Παράδειγμα 1

Αν ο χρήστης του προγράμματος δώσει:

```
python commentz_walter.py 'cacbaa' 'acb' 'aba' 'acbab' 'ccbab' example.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο `example.txt` και θα εμφανίσει:

```
acb: 0
acbab: 0
aba: 3
acb: 7
cacbaa: 6
```

Παράδειγμα 2

Αν ο χρήστης του προγράμματος δώσει:

```
python commentz_walter.py -v 'cacbaa' 'acb' 'aba' 'acbab' 'ccbab' example.txt
```

τότε το πρόγραμμα θα διαβάσει το αρχείο `example.txt` και θα εμφανίσει στην έξοδο:

```
0: 1,3
1: 1,2
2: 3,2
3: 3,2
4: 3,2
5: 3,2
6: 3,2
7: 1,3
8: 2,3
9: 2,2
10: 1,2
11: 3,2
12: 1,1
13: 3,1
14: 3,1
15: 3,1
16: 3,1
acb: 0
acbab: 0
aba: 3
acb: 7
cacbaa: 6
```

Περισσότερες Πληροφορίες

Ο αλγόριθμος Commentz-Walter επινοήθηκε από την Beate Commentz-Walter (1979a, 1979b). Όπως είδαμε υιοθετεί τις ιδέες του αλγόριθμου Boyer-Moore-Horspool (Horspool 1980) συνδυάζοντάς τες με έναν άλλο αλγόριθμο αναζήτησης συμβολοσειρών, τον αλγόριθμο Aho-Corasick (Aho και Corasick 1975), που εφευρέθηκε από τον Alfred V. Aho και την Margaret J. Corasick, ο οποίος χρησιμοποιεί trie στη λειτουργία του. Τα tries, όπως τα είδαμε, λειτουργούν ως *αυτόματα* (automata), τα οποία είναι από τα θεμέλια στη θεωρία υπολογισμού (theory of computation) και στις γλώσσες προγραμματισμού. Οι κανονικές εκφράσεις, για παράδειγμα,

βασίζονται σε αυτόματα. Για να μάθετε για αυτόν τον γοητευτικό κλάδο της πληροφορικής, δείτε το βιβλίο του Michael Sipser (2013).

References

Aho, Alfred V., and Margaret J. Corasick. 1975. "Efficient String Matching: An Aid to Bibliographic Search". *Communications of the ACM* (New York, NY, USA) 18 (6): 333–340. ISSN: 0001-0782.

Commentz-Walter, Beate. 1979a. *A string matching algorithm fast on the average.* Technical report TR 79.09.007. Heidelberg Scientific Center: IBM Germany.

———. 1979b. "A string matching algorithm fast on the average". In *Automata, Languages and Programming,* edited by Hermann A. Maurer, 118–132. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-35168-9.

Horspool, R. Nigel. 1980. "Practical fast searching in strings". *Software: Practice and Experience* 10 (6): 501–506. ISSN: 1097-024X.

Sipser, Michael. 2013. *Introduction to the Theory of Computation.* 3rd. Boston, MA: Cengage Learning.

Καλή Επιτυχία!