Tous les tests qui suivent ont été simulés en performant 20 itérations de chaque méthode. Les résultats sont en secondes, et chaque test se base sur une recherche effectuée sur le document correspondant dans le fichier Benchmark prototype. Les documents utilisés pour les tests 1 à 6 comprennent chacun 20 expressions régulières. J'ai effectué les recherches sur le fichier de 2.5 Mb puisque c'était le plus gros que j'étais en mesure de traiter (avec l'annotation récursive de BitVector) avec mon ordinateur personnel (8GB de mémoire). J'ai inclus mes commentaires et réflexions par rapport aux résultats à la fin du document.

Test préliminaire: CFTC.txt

Méthode	Suffix trees-NFA	Algorithme standard
matches()	73.307	1.986
count()	73.731	164.901
findAllSeq()	75.037	164.763

Test 1: Solide

Méthode	Suffix trees-NFA	Algorithme standard
matches()	0.795	4.812
count()	0.784	5.613
findAllSeq()	0.816	5.325

Test 2: CharacterClass

Méthode	Suffix trees-NFA	Algorithme standard
matches()	156.021	0.031
count()	158.878	21.666
findAllSeq()	157.863	23.231

Test 3: CharacterClass + don’t care/CharacterRange

Méthode	Suffix trees-NFA	Algorithme standard
matches()	11.524	17.027
count()	11.902	18.595
findAllSeq()	11.805	18.448

Test 4: Solide +range

Méthode	Suffix trees-NFA	Algorithme standard
matches()	1.08	13.686
count()	1.078	11.665
findAllSeq()	1.009	11.902

Test 5: Kleene closure (*,+)

Méthode	Suffix trees-NFA	Algorithme standard
matches()	2.028	24.843
count()	2.039	23.993
findAllSeq()	2.032	24.151

Test 6: Simulation finale (syntaxe mélangée)

Méthode	Suffix trees-NFA	Algorithme standard
matches()	160.722	37.38
count()	163.318	48.823
findAllSeq()	163.475	49.731

Commentaires

Avant tout, je veux simplement noter que la méthode matches() est de loin la moins performante relativement à l'algorithme standard dû au fait qu'une traversée de l'arbre doit tout de même se faire, alors qu'en pratique les méthodes standard de java trouvent une première occurrence dans le texte assez rapidement.

Malgré les résultats décevants, je crois qu'il y a beaucoup de possibilités d'amélioration. Dans la version antérieure de ma simulation, plus susceptible à l'erreur, le programme semblait déjà s'exécuter jusqu'à 6 fois plus rapidement que ce qui est indiqué ci-haut. J'ai préféré utiliser la version plus propre pour les tests, mais ça démontre qu'il y a certainement des possibilités pour accélérer le programme.

D'abord, un des problèmes majeurs avec l'implémentation actuelle est la création d'objets exponentielle que j'utilise pour vérifier chaque branche dans l'arbre de suffixe. C'est d'ailleurs ce qui mène à la mauvaise performance pour les tests impliquant beaucoup de ''CharClass''. En effet, une annotation préliminaire de l'arbre, ou bien un arbre construit avec des noeuds ayant des références à leurs transitions pourrait, à mon avis, accélérer la simulation en réduisant le nombre d'opérations pour déterminer si une branche sortante d'un noeud constitue bel et bien un chemin valide.

Par ailleurs, je crois qu'il serait bénéfique de laisser faire l'étape intermédiaire de transformer les expressions régulières en expressions postfixes afin d'effectuer la construction de Thompson. J'opterais plutôt pour une construction inspirée de Glushkov, construisant directement le NFA à partir d'un ''parsing tree''. La construction a une complexité plus importante que celle de Thompson, mais elle possède l'avantage majeure de ne générer aucune transition nulle, ainsi qu'un nombre constant d'états. Ceci faciliterait la représentation du NFA sous-forme de tableau d'entiers et pourrait grandement accélérer la simulation. À mon avis, ce changement pourrait drastiquement augmenter l'efficacité du programme, puisque l'implémentation actuelle avec énormément de pointeurs entre objets est trop lente.