Analyse en Implementatie van Drie Trie Algoritmen

Elias Nijs

Algoritmen en Datastructuren 3
Universiteit Gent

ABSTRACT

We analyseren drie theoretische algoritmen voor het opslaan en zoeken van multi-key data, en implementeren voor elke van deze een praktische c implementatie waarbij de sleutels ASCII strings zijn. Het eerste algoritme is een arraytrie, meer bepaald de patricia-trie. Het tweede algoritme is een combinatie van tries en binaire zoekbomen, de ternary-trie. Het laatste algoritme is een combinatie van ternary tries en binaire semi-splay trees. Tot slot zullen we deze 3 soorten tries met elkaar vergelijken op vlak van snelheid en geheugengebruik.

14 December 2022

Analyse en Implementatie van Drie Trie Algoritmen

Elias Nijs

Algoritmen en Datastructuren 3
Universiteit Gent

1. Introductie

Een trie, ook bekend als een digital tree of prefix tree, is een datastructuur die gebruikt wordt voor het opslaan van multi-key data. Deze data bestaat meestal uit verscheidene ASCII strings. Tries worden vaak gebruikt voor het implementeren van dictionaries waar snelle opzoek, toevoeg en verwijder operaties cruciaal zijn.

Over tijd zijn er verschillende soorten tries ontwikkeld, elk met hun eigen eigenschappen en afwegingen. In dit verslag bespreken we 3 verschillende soorten tries: de patricia trie[1] , de ternary trie[2] en tot slot een semi-splay ternary trie.

De patricia trie is een geheugen-efficiente trie. Deze efficientie wordt bereikt door enkel nodes voor vertakking op te slaan in plaats van voor alle karakters.

De ternary trie is een trie waarbij elk mogelijk volgend karakter opgeslaan wordt in een binaire boom. Daarboven op bevat elke node ook nog een pointer die naar de volgende binaire boom wijst. Op deze manier slaagt ook deze trie variant erin om heel geheugen efficient te zijn.

De laatste trie die we zullen bekijken, de semisplay ternary trie, is een variant van de ternary trie waarin de semisplay operatie verwerkt is. Het idee achter deze boom is om meer voorkomende prefixen dicther bij de wortel te brengen en zo zoekoperaties te versnellen. In de analyse zal blijken of de tijd die de splay operatie inneemt opweegt tegenover de gewonnen tijd die de herbalancering ons geeft.

In dit verslag zullen we de eigenschappen, implementaties en efficientie van deze tries in detail bespreken. Daarnaast zullen we ook de verschillende tries op vlak van geheugen gebruik en snelheid vergelijken met elkaar.

2. Algoritmes en hun Implementaties

In deze paragraaf zullen we de verschillende eigenschappen en implementaties van de drie onderzochte tries bespreken.

2.1. PATRICIA

PATRICIA, kort voor *Practical Algorithm To Retrieve Information Coded in Alphanumeric* , is een trie structuur ontwikkeld door Donald R. Morrison in 1968. Deze trie is een geheugen efficiente variant op de traditionele trie structuur. Het idee achter deze efficientie is om enkel vertakkingen op te slaan en geen lange paden waardoor het aantal nodes in de boom aanzienelijk daalt.

2.1.1. Overslaan van karakters

Het overslaan van karakters is het cruciale idee achter de patricia trie en waarom deze zo geheugen efficient is. Dit wordt bereikt door elke node een extra variable *skip* te laten bijhouden die zegt hoeveel lettertekens er zonder vertakking zijn.

Bij het invoegen van een nieuwe key kan het natuurlijk gebeuren dat op het pad dat we afgaan er een verschil is binnen de overgeslagen karakters van de node waar we op dat moment zitten. Op dat moment is het noodzakelijk deze node in twee te splitsen om een vertakking toe te voegen.

2.1.2. Geheugen vs Snelheid

Omwille van hoe ingewikkeld de invoeg operatie is bij patricia tries zullen we, hoewel we geheugen gebruik besparen, veel tijd verliezen bij deze operatie. We kunnen dit echter oplossen door een beetje geheugen gebruik op te offeren en met dit geheugen de overgeslagen deelstrings bij te houden. Op deze manier kunnen we tijdens een opzoek operatie al de prefixen vergelijken en hoeven we niet telkens tot aan een blad te lopen.

2.1.3. Implementatie

De implemenatie kan teruggevonden worden onder `src/arraytrie.c`.

We bespreken nu kort enkele implementatie details.

We beginnen met de structuur van onze implementatie. Deze ziet er als volgt uit:

```
struct atrie_node {
     char *ls;
     char *s;
     int16 m;
     struct atrie_node
          *next[ARRAYTRIE_ALPHASIZE];
};

struct atrie {
     struct atrie_node *root;
     size_t wc;
};
```

We hebben een wrapper *atrie* voor onze trie, deze houdt de wortel bij en hoeveel woorden er aanwezig zijn. Dit laatste werd gedaan zodanig dat het opvragen van de grootte van de trie aanzienelijk vereenvoudigd wordt.

Verder hebben we ook de structuur *atrie_node* voor onze nodes. Deze houdt een aantal verschillende zaken bij. Ten eerste houdt deze een array met pointers bij. Elk van deze pointers zal naar een volgende node wijzen indien deze een vervolg vormt op de huidige prefix. Verder houden we ook s en m bij. Zoals in de vorige paragraaf besproken werd, kunnen we onze operaties versnellen door de deelstring in elke node bij te houden. In deze implementatie werd ervoor gekozen dit te gebruiken. s is een pointer naar de deelstring en m is de lengte van deze deelstring. Tot slot hebben we ook nog *ls*. Dit is de pointer naar de finale string in een blad. Indien de node geen blad is, zal deze een null-pointer zijn. Op deze manier kunnen we deze pointer dus ook gebruiken om te checken of we in een blad zitten. Eventueel zouden we deze laatste variable nog kunnen weglaten en onze string op bouwen tijdens de zoekoperatie. Op deze manier zouden we nog extra geheugen kunnen besparen.

Onze implementatie ondersteunt alle verwachte operaties:

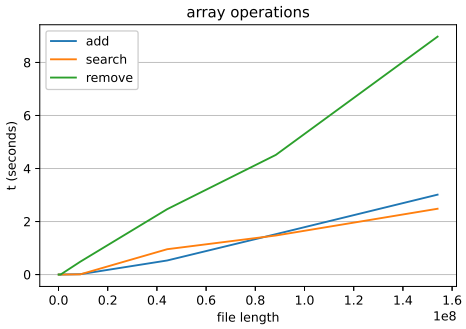
- `init: () -> atrie`
- `free: atrie -> ()`
- `search: (atrie, string) -> bool`
- `add: (atrie, string) -> bool`
- `remove:(atrie, string) -> bool`
- `size: atrie -> size_t`

De implementaties van al deze zijn relatief straight-forward en worden hier dus niet in detail bekeken.

2.1.4. Complexiteit

De complexiteit van deze boom is $O(m)$.

We kunnen dit ook verifieren aan de hand van een kleine benchmark.



Een grotere analyse van de performantie volgt later.

2.2. Ternary Trie

Ternary tries zijn een trie structuur beschreven door J. Bentley en B.Sedgewick in 1988. Deze structuur is een geheugen efficiente variant op de traditionele trie. Het idee achter deze trie soort is om een volgend karakter voor te stellen als een binaire boom die alle mogelijk volgende karakters bevat en daar bovenop elke node een extra kind te geven die naar de volgende binaire boom wijst. Omwille van de snelheid waarmee deze trie prefixen kan opzoeken, wordt hij vaak gebruikt voor programmas zoals autocomplete.

2.2.1. Staart compressie

Om te voorkomen dat we op het einde lange staarten hebben zonder vertakking, wordt er vaak geopteerd om de staart te comprimeren naar één node. We kunnen dit op twee manieren doen. De eerste manier is om telkens de volledige key op het einde op te slaan en de tweede manier is om de resterende karakters van de key op het einde op

te slaan. Welke van deze twee er gebruikt wordt is een afweging tussen geheugen gebruik en snelheid.

2.2.2. Implementatie

De code voor deze trie val terug te vinden onder `src/ternarytrie.c`.

We beginnen met naar de structuur te kijken.

```
struct ttrie_node  {
     char *s;
     char splitchar;
     union {
          struct ttrie_node *cs[3];
          struct {
               struct ttrie_node
                    *lo, *hi, *eq;
          };
     };
};
struct ttrie {
     struct ttrie_node *root;
     size_t wc;
};
```

We hebben een wrapper *ttrie* voor onze trie, deze houdt de wortel bij en hoeveel woorden er aanwezig zijn. Dit laatste werd gedaan zodanig dat het opvragen van de grootte van de trie aanzienelijk vereenvoudigd wordt.

Daarnaast hebben we ook de structuur *ttrie_node* die een node representeert. Deze node houdt een aantal zaken bij. Het eerste dat deze node bijhoudt zijn de pointers naar de volgende nodes. In onze structuur kunnen we deze op twee manieren addresseren. We kunnen ten eerste deze addresseren via de array *cs* en ten tweede met hun naam, *lo, hi, eq*. Deze eerste manier was origineel niet aanwezig, maar werd later toegevoegd zodat het addresseren van kinderen via een vergelijking mogelijk werd. Zo kunnen we bijvoorbeeld het volgende doen.

```
n = n->cs[c > n->splitchar]
```

In dit voorbeeld zetten we n gelijk aan een van de kinderen van n. Indien c groter is dan de splitchar van n, dan zal n gelijk gesteld worden aan `n->hi` en anders aan `n->lo`.

Verder bevat een node ook nog de splitchar. Dit is het karakter dat de node representeert. En tot slot bevat de node ook nog een pointer *s* naar een string. Deze string zal wijzen naar de volledige key indien de node een blad is en anders zal deze een null-pointer zijn. Op deze manier kunnen we *s* gebruiken om te checken of de node een blad is.

Net zoals bij de patricia trie werden de volgende operaties geimplementeerd:

- `init: () -> ttrie`
- `free: ttrie -> ()`
- `search: (ttrie, string) -> bool`
- `add: (ttrie, string) -> bool`
- `remove:(ttrie, string) -> bool`
- `size: ttrie -> size_t`

De search en add operaties werden iteratief geimplementeerd en de remove, free en print operaties recursief. De implementaties van deze operaties zijn ook hier allemaal relatief straight-forward en worden dus niet in detail bekeken.

Tot slot werden er 2 custom allocators gemaakt. De eerste is een arena allocator en de andere een pooling allocator. De implementatie met uitleg over hoe deze werken valt terug te vinden onder

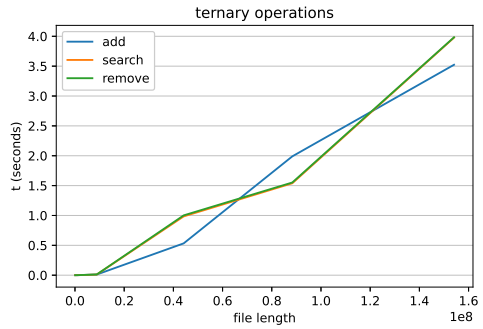
- `src/utils/m_arena.c`
- `src/utils/m_pool.c`

Deze zijn in staat om een signficante boost in performance te geven door het uitsparen van veel dure malloc/calloc operaties en doordat er minder geheugen fragmentatie optreedt. Helaas kunnen deze enkel gebruikt worden indien men een goede bovengrens kan bepalen voor hoeveel geheugen men nodig heeft. Om deze te gebruiken moet men immers op voorhand een grote buffer waar de allocators op geldig zijn reserveren. Een bijkomend voordeel is dat men de hele trie in één operatie kan vrijgeven door de backbuffer vrij te geven.

2.2.3. Complexiteit

De complexiteit van deze boom is $O(m)$.

We kunnen dit ook verifieren aan de hand van een kleine benchmark.



Een grotere analyse van de performantie volgt later.

2.3. Semisplay Ternary Trie

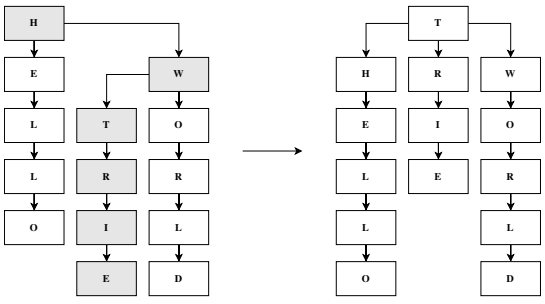
De semisplay ternary trie is een variant op de ternary trie waarin de semisplay operatie verwerkt is. Het idee achter deze trie is om aan de hand van de semisplay operatie vaak voorkomende prefixen naar boven te splayen. Op deze manier zullen we een beter gebalanceerde structuur krijgen en zal het opzoeken van vaak voorkomende woorden sneller verlopen.

Deze trie is mijn custom trie. Of deze performant is en of de trie sneller is dan een normale ternary trie zal moeten blijken in de analyse die later volgt.

2.3.1. De semisplay operatie

De semisplay operatie bestaat erin het gevolgde pad te semisplayen. Meer bepaalde zal dit bij een ternary trie beteken dat elke binarie boom die op ons pad ligt zal gesemisplayed worden.

Dit is beter te zien aan het volgende voorbeeld. In dit voorbeeld zoeken we de key *trie* en zien we hoe de trie zich daarna balanceert.



2.3.2. Implementatie

De code voor deze trie val terug te vinden onder `src/customtrie.c`.

2.3.2.1. Structuur

We beginnen terug met eerst naar de structuur te kijken.

```
struct ctrie_splay_node {
     struct ctrie_splay_node *next;
     struct ctrie_node *node;
};

struct ctrie_node  {
     char *s;
     char splitchar;
     union {
          struct ctrie_node *cs[3];
          struct {
               struct ctrie_node
                    *lo, *hi, *eq;
          };
     };
};

struct ctrie {
     struct ctrie_node *root;
     size_t wc;
     size_t splaycnt;
     struct ctrie_splay_node *splayhead;
     struct ctrie_node **splayroot;
};
```

De structuur van onze implementatie bestaat uit drie delen: een wrapper, *ctrie*, een node uit de trie, *ctrie_node*, en als laatste een node uit de linked list die het play pad bijhoudt, *ctrie_splay_node*.

We kijken eerst naar de wrapper. De wrapper bevat de wortel van de trie en hoeveel woorden er aanwezig zijn net zoals in de implementatie van de ternary trie. Naast deze variablen bevat de wrapper ook data omtrent het splay gedeelte. Deze data bestaat ten eerste uit een splayhead. Deze pointer wijst naar de eerste node van de linked list die het splaypad bijhoudt. Verder hebben we ook nog de splaycount, deze houdt bij hoeveel nodes er in de lijst zitten. Tot slot hebben we nog de splayroot, deze pointer wijst naar de pointer die naar de root van de laast gepasseerde binaire boom wijst. Deze laatste variable zorgt ervoor dat we de laatste stap van een splay operatie correct kunnen uitvoeren. Het kan immers zijn dat de binarie boom een nieuwe wortel krijgt. Deze moete we dan correct terug in de trie hangen aan de hand van deze splayroot.

De structuur van *ctrie_node* is exact dezelfde als deze van een node in de ternary trie.

Als laatste hebben we nog de *ctrie_splay_node*. Deze structuur representeert een node in een linked list. Hij bevat een pointer naar de data en naar de volgende node.

2.3.2.2. Operaties

Net zoals bij de ternary trie werden de volgende operaties geimplementeerd:

- `init: () -> ttrie`
- `free: ttrie -> ()`
- `search: (ttrie, string) -> bool`
- `add: (ttrie, string) -> bool`
- `remove:(ttrie, string) -> bool`
- `size: ttrie -> size_t`

Naast deze operaties zijn er ook een aantal interne operaties om de semisplay operatie te verrichten.

We hebben volgende functies voor de operaties op de lijst die ons splaypad bijdhoudt. Deze lijst gedraagt zich als een lifo.

- `splay_push: (ctrie, ctrie_node)->()`
- `splay_pop: ctrie -> ctrie_node`
- `splay_clear: ctrie -> ()`

Tot slot hebben we de splay operatie zelf.

- `splay: ctrie -> ()`

De implementatie van deze operaties is relatief straight-forward. Deze worden dan ook niet in detail besproken.

2.3.3. Complexiteit

De complexiteit van deze boom is $O(log(n))$.

Dit komt doordat de trie constant geherbalanceerd wordt. Het aantal nodes dat we moeten doorlopen zal dus nooit zeer extreem groeien.

3. Analyse

In deze paragraaf zullen verschillende vergelijkingen maken op vlak van zowel snelheid als geheugengebruik.

Deze analyses werden uitgevoerd op een reeks bestanden gegenereerd aan de hand van blok-circulante Ramsey grafen.

Deze bestanden zijn de volgende:

Naam	Aantal woorden
geschud_piepklein.g6	1000
geschud_klein.g6	10000
geschud_middelmaat.g6	100000
geschud_groot.g6	500000
geschud_heelgroot.g6	1000000
geschud.g6	1743797

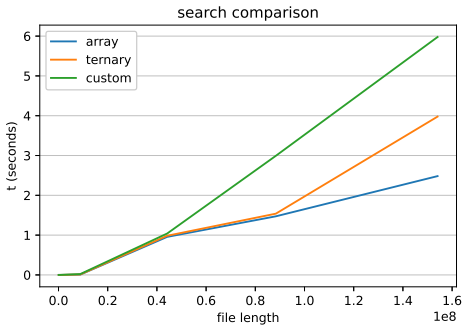
3.1. Tijd analyse

3.1.1. Analyse operaties

We beginnen met een analyse van de alleenstaande operaties. Voor elk van deze werd door alle woorden gelopen en telkens de operatie toegepast.

3.1.1.1. Zoeken

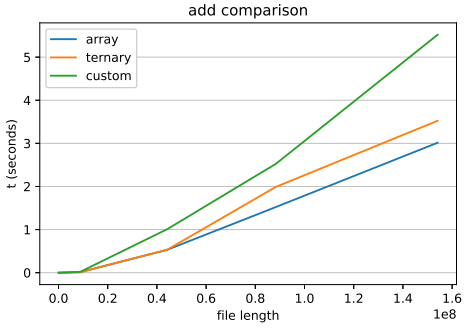
Deze grafiek toont de snelheid van de verschillende tries bij zoek operaties.



We zien dat de patricia trie hier het best presteert en de semisplay ternary het slechtst.

3.1.1.2. Toevoegen

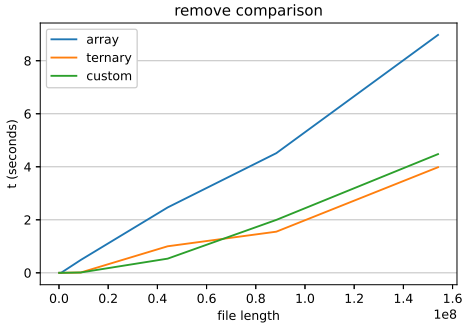
Deze grafiek toont de snelheid van de verschillende tries bij toevoeg operaties.



We zien ook hier dat de patricia trie het best presteert en de semisplay ternary het slechtst.

3.1.1.3. Verwijderen

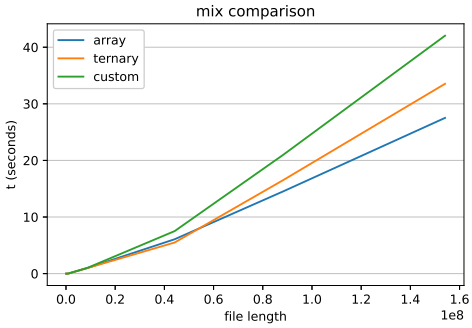
Deze grafiek toont de snelheid van de verschillende tries bij verwijder operaties.



We zien dat de ternary trie het best presteert en de patricia trie het slechtst presteert.

3.1.2. Globale analyse

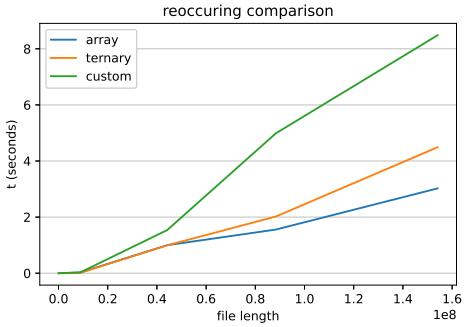
Verder werd ook een analyse op een mix van operaties gedaan om een beter globaal beeld te krijgen.



Hier zien we dat de patricia trie het best presteert en de semisplay ternary het slechtst.

3.1.3. Terugkomende prefixen

Tot slot kijken we specifiek naar de snelheid van de verschillende tries bij vaak terugkomende woorden. Om dit te testen voegen we eerst alle woorden toe en zoeken vervolgens een aantal keer hetzelfde woorden op.



In dit geval zouden we verwachten dat de semisplay ternary het best presteert vermits deze het woord dichter bij de wortel zal brengen. Dit is echter niet geval. Integendeel, de ternary semisplay presteert het slechtst.

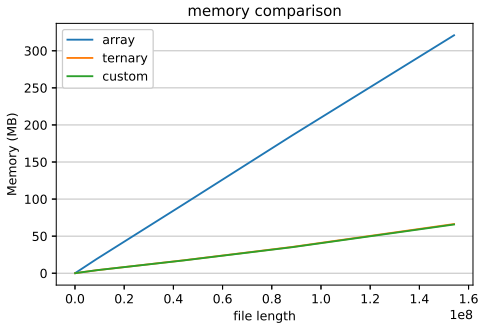
3.1.4. Tijd conclusie

We besluiten dat in het geval van weinig verwijder operaties men op vlak van snelheid het best voor de patricia trie kiest. Indien men wel veel verwijder operaties nodig heeft, is het voordeliger om voor de ternary trie te kiezen. In het algemeen kunnen we ook besluiten dat de semisplay ternary het slechtst presteert. Helaas blijkt dat bij de semisplay ternary de tijd die we winnen door de trie te balanceren niet opweegt tegenover de tijd die we verliezen met de semisplay operatie zelf. Dit is opmerkelijk vermits de tijd complexiteit van deze trie $O(log(n))$ is terwijl deze bij de andere twee $O(n)$ is.

3.2. Geheugen verbruik

Om het geheugen gebruik te vergelijken voegen we alle woorden toe en kijken naar het peakgeheugen verbruik. Merk op dat dit geen absolute benchmark is. Om het geheugengebruik te gaan meten maken immers gebruik van valgrind. Valgrind geeft echter het totale gebruik terug. Dit bevat dus ook bijvoorbeeld de data van het bestand die de woorden bevat.

We krijgen het volgende resultaat.



Hier zien we dat de ternary en semisplay ternary ongeveer gelijk lopen. De semisplay ternary verbruikt een heel klein beetje minder memory. De patricia trie daarentegen verbruikt aanzienelijk veel meer geheugen.

3.2.1. Geheugen conclusie

Indien men zoveel mogelijk geheugen wil uitsparen, gebruikt men het best de semisplay ternary trie. Dit is opmerkelijk aangezien deze in principe meer data moet opslaan.

4. Slotnoot

In conclusie, ternary, patricia en semisplay ternary tries zijn alle drie datastructuren voor het efficient opslaan en opvragen van multikey data. We hadden verwacht de de semisplay ternary voordelig zou zijn wanneer woorden vaak terugkomen. Dit bleek echter niet het geval te zijn.

References

1. Morrison, Donald R, “PATRICIA - practical algorithm to retrieve information coded in alphanumeric,” *Journal of the ACM (JACM)* **15**(4), pp. 514-534, ACM New York, NY, USA (1968).
2. Bentley, Jon L and Sedgewick, Robert, “Fast algorithms for sorting and searching strings” in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms,* pp. 360-369 (1997).