Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Vít Urbánek

Reprezentace řetězců v hašovacích tabulkách

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Alena Koubková, CSc.
Studijní program: Informatika, Softwarové systémy

2008

Rád bych poděkoval RNDr. Aleně Koubkové, CSc., za cenné rady poskytnuté v průběhu tvorby této práce a čas, který vedení této práce věnovala. Také bych rád poděkoval svým rodičům za podporu během celého studia.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 14. dubna 2008 Vít Urbánek

Obsah

Název práce: Reprezentace řetězců v hašovacích tabulkách
Autor: Vít Urbánek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Alena Koubková, CSc.
e-mail vedoucího: koubkova@ksi.ms.mff.cuni.cz

Abstrakt: Základním problémem hašování je řešení kolizí. Jednou z možností řešení tohoto problému je vytváření řetězců kolidujících prvků. Řetězce se ukládají buď uvnitř, nebo vně tabulky a jsou obvykle reprezentovány jako neuspořádané spojové seznamy. Cílem této práce je navrhnout alternativní struktury (uspořádané řetězce, samoupravující seznamy, ...) pro reprezentaci kolidujících prvků, implementovat je do známých algoritmů a alespoň experimentálně zhodnotit jejich vliv na rychlost základních operací v hašovacích tabulkách.

Klíčová slova: hašovací tabulka, samoupravující seznamy, experimentální analýza

Title: Representation of chains in hash tables
Author: Vít Urbánek
Department: Department of Software Engineering
Supervisor: RNDr. Alena Koubková, CSc.
Supervisor's e-mail address: koubkova@ksi.ms.mff.cuni.cz

Abstract: The essential problem of hashing is a solving of collisions of elements. One of possible solutions of this problem is chaining of colliding elements. The chains are stored inside or outside the table and they are usually represented as unsorted linear lists. The aim of this thesis is to design some alternative structures (sorted linear lists, self-organizing linear lists, etc.) for representation of colliding elements, to implement them into known algorithms and experimentally evaluate their effect on efficiency of dictionary operations (Insert, Member, Delete).

Keywords: hash table, self-organizing lists, experimental analysis

Kapitola 1

Úvod

Jednou z možností, jak uchovávat data, je využít hašovacích tabulek. Jejich cílem je umožnit provádění operací v konstantním čase v průměrném případě při udržení rozumné velikosti reprezentace. V hašování ale může docházet ke kolizím (v tabulce jsou prvky se stejnou hašovací adresou) a kolidující prvky bývají poté ve většině hašovacích metod udržovány v podobě jakýchsi řetězců. Tyto řetězce jsou někdy udržovány mimo tabulku, někdy přímo v ní a mívají podobu nesetříděných spojových seznamů. Navíc platí, že po vložení prvků se už pozice prvků v řetězci nemění, pouze mohou přibýt nové prvky (v některých metodách jen na konec řetězce, v jiných i jinam). Právě tyto tři vlastnosti – neměnnost pořadí, nesetříděnost a že se jedná o seznam – nás inspirovaly k třem základním možným modifikacím obvyklé reprezentace řetězců, kterými jsou samoupravující seznamy, setříděné seznamy a stromy.

Motivací pro používání samoupravujících seznamů je měnit podobu seznamu tak, abychom v řetězci museli procházet co nejméně prvků a tím aby všechny operace byly co nejrychlejší. Problému udržování nesetříděného seznamu, které zahrnuje jeho různé možné úpravy, se říká list-update problém. Jako první se jím zabýval McCabe [14], který navrhl dvě konkrétní samoupravující metody (strategie), Move-To-Front a Transpose, které byly i v pozdější době velmi hojně zkoumány. Výzkum kolem list-update problému se nejprve zaměřil na studium samoupravujících metod za předpokladu, že série požadavků (typicky argumenty operace vyhledání), které zpracovávají, jsou generovány podle nějakého pravděpodobnostního rozdělení. Sleator a Tarjan [20] byli později první, kteří se zabývali samoupravujícími algoritmy z hlediska kompetitivní analýzy a velmi ji zpopularizovali. V devadesátých letech se objevily i první randomizované algoritmy pro list-update problém, které právě z hlediska kompetitivity překonaly optimální deterministické.

Samoupravující strategie byly v minulosti většinou zkoumány jen pro samotný seznam. V různých článcích sice bylo zmiňováno, že by bylo možné použít samoupravující strategii v rámci hašování, ale, pokud víme, nikdo se tomu dále podrobněji nevěnoval. Proto se všechny dostupné teoretické výsledky týkají jen jejich použití pro seznam a o jejich praktickém využití v hašování se nám také podařilo najít jen jeden článek – [24].

V této práci jsme se rozhodli vybrat pro další zkoumání tři samoupravující algoritmy, a to již výše zmíněné Move-To-Front a Transpose a pak jeden randomizovaný algoritmus, Timestamp(p).

Prozkoumat chování samoupravujících algoritmů v hašování asi bylo největší motivací v této práci, ale jak už bylo řečeno, byly zkoumány ještě další dvě alternativní možnosti reprezentace řetězců.

Metoda setříděných řetězců je (alespoň v případě vyhledávání podle rovnoměrného rozdělení) stejně dobrá při úspěšném vyhledání jako původní metoda, ale nabízí rychlejší vkládání a neúspěšné vyhledání.

Poslední zkoumanou možností je nahradit seznam stromem. V našem případě jsme použili klasický binární vyhledávací strom. Přestože se neprovádí žádné vyvažování, tak jeho výška je při náhodném pořadí vkládaných prvků logaritmická, a tak pro počty prvků v řádu stovek a výš je jeho použití daleko rychlejší než použití libovolného seznamu. V hašování se ale často vyskytují ne stovky, ale třeba jen několik kolidujících prvků a zajímavé bylo právě zkusit, jak na tom budou stromy ve srovnání se seznamy v takovéto situaci.

Pokud jsme o něco výše mluvili o samoupravujících metodách a nyní o stromech, je dobré připomenout, že existují i stromy používající samoupravování (těmi nejznámějšími jsou zřejmě Splay stromy). Jimi jsme se ale v této práci nezabývali.

Co se týče hašovacích metod, tak jsme se omezili na jednu hašovací metodu, kterou je metoda separovaných řetězců. Jedná se o jednu ze základních hašovacích metod, jejíž princip je velmi jednoduchý. Kolidující prvky jsou uloženy mimo tabulku a v tabulce na místě odpovídajícím jejich společné adrese je jen odkaz na první prvek (existují i verze, kdy je tam uložen první prvek). Do této hašovací metody byly implementovány uvedené alternativní možnosti reprezentace řetězců a zjišťovalo se, jaký bude mít jejich použití vliv na rychlost slovníkových operací jako Insert, Member, Delete.

Práce se původně měla týkat, jak její název napovídá, pouze hašovacích tabulek, ale v průběhu práce se ukázalo jako zajímavé provést srovnání použitých metod i na samotných seznamech. Testované délky těchto seznamů se pohybovaly asi od 500 do 10000. Mohlo by se zdát, že testování seznamů takovýchto délek je vzhledem k použití v hašování irelevantní, ale jak bylo ukázáno v [24], použití hašovací tabulky (s metodou separovaných řetězců) s průměrnou délkou řetězců skoro 600 může být přibližně stejně výhodné jako použití binárního vyhledávacího stromu, a při použití strategie Move-To-Front dokonce skoro dvakrát rychlejší. Proto má určitě smysl zkoumat i seznamy delší než jen několik desítek prvků, a protože testování hodně dlouhých řetězců v hašovací tabulce by znamenalo nutnost použití obrovského počtu prvků, tak je jednodušší, i když ne úplně přesné, chování odhadnout z experimentů na samotném seznamu. U testů prováděných na hašovacích tabulkách už jsme pak použili průměrné délky řetězců v řádu jednotek až desítek a sledovali rozdíly v chování jednotlivých strategií při testech na seznamu a na hašovací tabulce.

V kapitole 2 se věnujeme hašování, zejména metodě separovaných řetězců, která je pak dále používána. Kapitola 3 rozebírá alternativní metody reprezentace řetězců. Kapitola 4 se zaměřuje na samoupravující strategie, především na způsoby porovnávání, jak je která samoupravující metoda dobrá, a na výsledky zjištěné v této oblasti. Dále jsou pak také popsány výsledky některých experimentů z minulosti, které se zabývaly samoupravujícími metodami. Kapitola 5 je věnována provádění experimentů. Jsou zde popsána použitá pravděpodobnostní rozdělení, generování dat, testované operace, zjišťované údaje, způsob měření času. V kapitolách 6 a 7 se zabýváme výsledky experimentů prováděných na seznamech, resp. tabulkách. Kapitola 8 shrnuje výsledky, ke kterým se v této práci dospělo, její přínos a ukazuje možnosti pro další práci.

Kapitola 2

Hašovaní

Velmi častým problémem, který se v praxi řeší, je uchovávání dané množiny záznamů, vyhledávání v ní, přidávání nových záznamů a mazání nepotřebných. Každý takový záznam má obvykle klíč, který ho jednoznačně identifikuje. Uložení klíče a samotného záznamu si můžeme představit odděleně. U uloženého klíče může být odkaz na uložení záznamu, a tedy při získání místa s klíčem už se na samotný záznam dostaneme snadno. V následujícím textu budeme proto uvažovat pouze problém reprezentace množiny klíčů záznamů a už nebudeme mluvit o odkazu na záznam, i když ho tam vždy budeme mlčky předpokládat.

Uvažujme univerzum klíčů U a množinu klíčů $M \subseteq U$, kterou chceme reprezentovat (z tohoto důvodu budeme pro klíč také používat výraz prvek). Klíče mohou být uloženy v poli a samotný klíč (pokud je číselný) může sloužit jako index do tohoto pole. To nám zajišťuje, že k danému prvku můžeme přistupovat v čase O(1). Problémem ovšem je to, že pole by muselo být tak velké, jaký je možný rozsah klíčů. Tento problém právě řeší hašování, a to tím způsobem, že místo klíče se jako index použije hašovací hodnota klíče, která je výrazně menší, a pole díky tomu může být o dost menší. Formálně řečeno hašovací funkce vypadá takto: $h : U \rightarrow 0, 1, ..., m-1$, kde m je velikost tabulky (pole) a funguje tak, že prvek $x \in S$ bude v tabulce (poli) vložen na pozici $h(x)$. Často se stává, že pro dva různé prvky $x, y \in S$ platí $h(x) = h(y)$. Tomuto jevu se říká kolize a řešení kolizí je základním problémem hašování.

V následujících dvou kapitolách budou popsány některé základní metody hašování. Zvláštní pozornost bude věnována metodě separovaných řetězců, protože ta je předmětem dalšího zkoumání v této práci.

2.1 Hašování se separovanými řetězci

Myšlenka hašování se separovanými řetězci je taková, že všechny navzájem kolidující prvky (prvky se stejnou hodnotou hašovací funkce) budou umístěny ve spojovém seznamu přidruženém k dané pozici tabulky, konkrétně tedy všechny prvky s hodnotou hašovací funkce k budou uloženy v seznamu umístěném na k-té pozici tabulky. Někdy bývá první prvek seznamu ukládán do tabulky, v této práci byl však použit přístup, kdy v tabulce je uložen pouze ukazatel na první prvek seznamu.

Poznámka. Pokud budeme mluvit o vkládání prvků do tabulky (nebo vyhledávání nebo mazání prvků v tabulce), budeme pod pojmem tabulka chápat kromě pole s ukazateli i přidružené seznamy s prvky.

Příklad
Nechť množina klíčů S reprezentovaná v hašovací tabulce je {15, 23, 45, 65, 128, 155, 168, 171} a hašovací funkce je $h(x) = x \ mod \ 10$. Označíme $R(k)$ množinu klíčů uložených ve spojovém seznamu, který je na k-tém místě tabulky. Potom $R(1) = \{171\}$, $R(3) = \{23\}$, $R(5) = \{15, 45, 65, 155\}$, $R(8) = \{128, 168\}$ a pro ostatní i je $R(i) = \{\}$.

Představme si, že hašovací tabulka obsahuje určitou množinu prvků S. V této situaci budeme chtít provádět tři základní operace:

- vložení prvku x (Insert(x))
- vyhledání prvku x (Member(x))
- smazání prvku x (Delete(x))

U každé z operací je prvním krokem výpočet $h(x)$, a tedy určení, do kterého seznamu prvek x patří. Při operaci Member(x) procházíme seznam, dokud nenarazíme na prvek x nebo na konec seznamu. V prvním případě se jedná o úspěšný Member (prvek x tam je), ve druhém o neúspěšný Member (prvek x tam není). Při operaci Insert(x) procházíme seznam, a pokud dojdeme až na jeho konec, aniž bychom narazili na prvek x, tak na konec seznamu prvek x vložíme. Operace Delete(x) je podobná jako Member(x), pouze po případném nalezení prvku následuje ještě jeho odstranění.

2.1.1 Složitost, odhady

V této části se budeme věnovat některým teoretickým výsledkům týkajícím se hašování se separovanými řetězci. V literatuře jsou uvedeny například v [22]. Zopakujme, že předpokládáme univerzum klíčů U, jeho podmnožinu S, kterou reprezentujeme v tabulce velikosti m, a hašovací funkci h. V následujících odhadech budeme předpokládat následující:

1. hašovací funkce h rozděluje univerzum U rovnoměrně, tedy $|h^{-1}(k) - h^{-1}(l)| \leq 1$ pro $k, l \ \in 0, 1, ..., m-1$
2. množina S je náhodně vybraná z univerza U tak, ze všechny množiny dané velikosti n mají stejnou pravděpodobnost výběru
3. každý prvek z U má stejnou pravděpodobnost, že bude argumentem dané operace

Očekávaná délka řetězce
Náročnost operace je závislá na délce příslušného seznamu (v případech, kdy tam argument operace není, je to jasné a brzy objasníme, že v průměrném případě to platí i pro případy, kdy tam argument operace je), proto je velmi užitečným údajem očekávaná délka seznamu. Označme n počet prvků v tabulce. Za výše uvedených předpokladů je očekávaná délka každého seznamu rovna n/m. Poměr $\alpha = n/m$ se také nazývá faktor naplnění.

Úspěšné a neúspěšné vyhledávání
Dále se budeme věnovat složitosti úspěšného a neúspěšného vyhledání. Jako měřítko složitosti použijeme počet testů, kde test bude porovnání klíčů nebo zjištění, že seznam je prázdný. V případě úspěšného vyhledání lze očekávaný počet testů aproximovat výrazem $1 + \alpha/2$, v případě neúspěšného výrazem $e^{-\alpha} + \alpha$.

2.2 Jiné možnosti

V této části představím stručně některé další metody hašovaní. Všechny uchovávají prvky uvnitř tabulky. Jejich společná myšlenka spočívá v tom, že pokud prvek nemůže být umístěn v tabulce na pozici, kam by patřil (tedy prvek x nemůže být umístěn na pozici $h(x)$), protože už je tam jiný prvek, bude umístěn na jinou pozici v tabulce. Navzájem kolidující prvky vytváří řetězec uvnitř tabulky a to, jak je určováno a uchováváno, kde by měl řetězec pokračovat, potom právě jednotlivé metody odlišuje.

Uvedené odhady složitosti jsou opět převzaty z [22]. Týkají se vždy jedné konkrétní modifikace algoritmu, u jiných modifikací mohou být výsledky nepatrně odlišné.

2.2.1 Hašování s přemísťováním

U této metody je řetězec uchováván jako obousměrný spojový seznam. Při určování volné pozice pro pokračování řetězce se postupuje náhodně, tedy vezme se libovolný volný řádek. Může se samozřejmě stát, že později na tento řádek budeme chtít vložit prvek, jehož hodnota hašovací funkce tomuto řádku odpovídá. V tom případě tento prvek dostane přednost a prvek, který zde byl původně, bude přesunut někam jinam (odtud název hašování s přemísťováním).

Každý řetězec navzájem kolidujících prvků tedy začíná na své správné hašovací adrese a každý řádek tabulky kromě klíče obsahuje ještě ukazatele na předchozí a následující prvek v řetězci, které se při vkládání a mazání aktualizují.

Složitost

Očekávaný počet testů u úspěšného vyhledání je stejný jako u metody separovaných řetězců, a lze ho tedy aproximovat výrazem $1 + \frac{\alpha}{2}$. Očekávaný počet testů u neúspěšného vyhledání lze odhadnout jako $1 + \frac{\alpha^2}{2}$.

2.2.2 Hašování se dvěma ukazateli

V této metodě je řetězec reprezentován jako jednosměrný seznam. Řetězec s prvky s hodnotou hašovací funkce i ale nemusí začínat na řádku i (ten může být již obsazen kolidujícím prvkem z jiného řetězce). To, kde opravdu začíná, je uvedeno v jedné z položek řádku i.

Složitost

Očekávaný počet testů u úspěšného vyhledání lze aproximovat výrazem $1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$. Očekávaný počet testů u neúspěšného vyhledání je stejný jako u hašovaní s přemísťováním, a lze ho tedy odhadnout jako $1 + \frac{\alpha^2}{2}$.

2.2.3 Srůstající hašování

V této metodě dochází k tomu, že v jednom řetězci se mohou vyskytovat prvky s různými hodnotami hašovací funkce. Pokud budeme vkládat prvek s hodnotou hašovací funkce i a i-tý řádek už bude obsazen (jedno jakým prvkem), tak nový prvek jednoduše vložíme do tohoto řetězce. Jednotlivé metody srůstajícího hašování se pak liší podle toho, na jaké místo do řetězce nový prvek zařadíme, a dále podle toho, jestli používají pomocnou paměť nebo ne.

Srůstající hašování bez pomocné paměti

- EISCH – nový prvek se přidává na začátek řetězce (prvek x se vloží hned za prvek na pozici $h(x)$)
- LISCH – nový prvek se přidává na konec řetězce

Srůstající hašování s pomocnou pamětí

V této variantě srůstajícího hašování existuje navíc ještě pomocná paměť, která není adresovatelná pomocí hašovací funkce. Pokud při vkládání nového prvku dojde ke kolizi, je pro nový prvek vybrán nějaký volný řádek z pomocné paměti, pokud je to možné. Až se pomocná paměť zaplní, začnou se volné řádky pro kolidující prvky brát z té části paměti, která je adresovatelná hašovací funkcí.

- EICH – nový prvek se přidává na začátek řetězce (prvek x se vloží hned za prvek na pozici $h(x)$)
- LICH – nový prvek se přidává na konec řetězce
- VICH – nový prvek x se přidává za poslední prvek řetězce, který je ještě v pomocné paměti (pokud tam žádný takový není, tak je nový prvek vložen do řetězce za prvek na pozici $h(x)$)

Složitost

Explicitní výsledky o složitosti těchto metod neuvádíme – vzorce jsou poměrně komplikované a kromě faktoru naplnění v nich vystupuje ještě tzv. adresový faktor, tj. poměr velikosti adresové a pomocné paměti. Tyto metody jsou ale obecně velmi rychlé i pro hodně zaplněné tabulky.

2.2.4 Dvojité hašování

U této metody se nevybírá volný řádek pro kolidující prvek libovolně, ale výběr je dán přesným předpisem. Kromě hašovací funkce h je dána ještě funkce $f : U \to 1, ..., m-1$, a pokud je při vložení prvku x obsazena pozice $h(x)$, zkusíme ho vložit na $h(x) + f(x)$ $mod\ m$ (m je velikost tabulky), pokud i tato pozice je obsazena, zkusíme ho vložit na $h(x) + 2 \cdot f(x)$ $mod\ m$ atd. Obdobně postupujeme i při vyhledávání. Díky tomuto předpisu není potřeba mít v tabulce položky pro organizaci spojového seznamu, a jsou tam tedy uloženy pouze klíče. Aby metoda správně fungovala, je nutné, aby $f(x)$ a m byly nesoudělné, a protože $f(x)$ může obecně nabývat všech hodnot od 1 do $m-1$, je obecně nutné, aby m bylo prvočíslo. Speciálním případem dvojitého hašování je lineární přidávání, kde $f(x) = 1$ a kde se tedy v případě kolize zkouší vždy bezprostředně následující řádek v tabulce. U lineárního přidávání není nutné, aby m bylo prvočíslo.

Složitost

V modelu dvojitého hašování lze očekávaný počet testů u úspěšného vyhledání aproximovat výrazem $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ a očekávaný počet testů u neúspěšného vyhledání výrazem $\frac{1}{1-\alpha}$. Pro lineární přidávání jsou známé odhady $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ pro úspěšné vyhledání a $\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$ pro neúspěšné vyhledání.

Tyto metody jsou v porovnání s předchozími v průměru pomalejší, a to především v hodně zaplněných tabulkách.

2.3 Reprezentace seznamů kolidujících prvků

Ve většině výše zmíněných metod byly kolidující prvky udržovány ve formě řetězců, reprezentovaných jako nesetříděné seznamy. V této práci se budeme zabývat alternativními reprezentacemi řetězců s tím, že se omezíme na metodu separovaných řetězců. Tato metoda totiž umožňuje vznik poměrně dlouhých seznamů, když reprezentovaná množina je velká vzhledem k velikosti základní tabulky. V takových případech mohou být operace na nesetříděných seznamech časově poměrně náročné. Budeme proto hledat jiné reprezentace umožňující rychlejší vyhledávání. Mezi zkoumané reprezentace budou patřit různé typy samoupravujících seznamů, setříděné seznamy a binární vyhledávací stromy. Budeme jednotlivé možné reprezentace řetězců implementovat do metody separovaných řetězců a porovnávat v jednotlivých případech (alespoň experimentálně) rychlost prováděných operací.

Kapitola 3

Zkoumané strategie

V této kapitole popíšeme zkoumané možnosti reprezentace řetězců a práce s nimi během operací. Tyto možnosti budeme nazývat strategie. Základní strategií, která už byla představena v kapitole 2.1, je reprezentace řetězců pomocí nesetříděných seznamů s tím, že se nedělají po vložení nebo vyhledání prvků žádné úpravy seznamu. Tuto strategii budeme nazývat klasickou. Nyní se budeme zabývat ostatními strategiemi.

3.1 Samoupravující strategie na seznamech

Podstatou samoupravujících strategií je to, že prvek, který je argumentem operace, je na konci operace přesunut (pokud tam je) blíže k začátku seznamu. Důvodem je, že předpokládáme, že by mohla být v budoucnu u tohoto prvku větší šance, že bude znovu vyhledáván (to samozřejmě nemusí být nakonec pravda). Jednotlivé samoupravující strategie se pak liší tím, za jakých podmínek a kam se bude prvek přesouvat.

3.1.1 Strategie Move-To-Front (MTF)

Strategie MTF představuje mezi samoupravujícími strategiemi jakýsi nejradikálnější přístup, protože prvek, který je argumentem operace, je na konci operace přesunut až na začátek seznamu. V případě úspěšné operace Insert je tedy nejprve vložen a pak přesunut, v případě neúspěšné dojde jen k přesunu, stejně jako v případě úspěšné operace Member. U neúspěšné operace Member nebo u operace Delete samozřejmě k žádnému přesunu nedochází, protože není co přesouvat.

3.1.2 Strategie Transpose (TR)

Strategie TR představuje jakýsi protipól ke strategii MTF, protože je na rozdíl od ní velice konzervativní a argument operace je přesunut pouze o jedno místo blíže k začátku seznamu (pokud už nebyl prvním prvkem seznamu). K uplatňování přesunu dochází ve stejných případech jako u strategie MTF.

3.1.3 Strategie Timestamp(p) – TS(p)

U této strategie opět může docházet k přesunům na konci operace (a je to opět ve stejných případech jako u MTF a TR), ale pravidla pro to, jestli přesouvat a kam, jsou v tomto případě o dost složitější, než tomu bylo v předchozích dvou případech. Tato strategie je randomizovaná, s pravděpodobností p se zvolí jedna podstrategie, s pravděpodobností $1 - p$ druhá. Parametr p může být jakékoliv reálné číslo z intervalu $[0, 1]$. Následuje přesný popis algoritmu.

Algoritmus TS(p)
Nechť je dána posloupnost požadavků (operací) $s = s(1), s(2), ..., s(m)$ (čísla v závorkách značí časové okamžiky provedení operací, nikoli jejich argument). Pak každý požadavek $s(t)$, $1 \leq t \leq m$, je zpracován následovně. Předpokládejme, že $s(t)$ je požadavek na prvek x (to znamená, že $s(t)$ je operace, jejímž argumentem je x). Pokud x bude v seznamu po provedení „klasické“ operace (tedy $s(t)$ je Insert nebo úspěšný Member), tak potom:

s pravděpodobností p se provede krok (a) – MTF
(a) Přesuň x na začátek seznamu

s pravděpodobností $1 - p$ se provede krok (b) – TS
(b) Pokud v intervalu $[1, t - 1]$ nebyl na prvek x žádný požadavek, ponech x v seznamu na jeho místě. Jinak nechť $t' \in [1, t - 1]$ je čas posledního předchozího požadavku na x a obsluž požadavek $s(t)$ následovně. Nechť $v_x(t)$ je prvek nejblíže začátku seznamu, který předchází x a
(i) nebyl na něj požadavek během intervalu $[t', t - 1]$
nebo
(ii) byl na něj právě jeden požadavek během intervalu $[t', t-1]$ a ten byl obsloužen za použití kroku (b).
Pokud tam takový prvek není, nechť $v_x(t) = x$. Přesuň x bezprostředně před $v_x(t)$ (pokud $v_x(t) = x$, tak nedělej nic).

Poznámka. Požadavky na prvek se začínají „počítat“ od jeho úspěšného vložení a při smazání se nulují. Pokud tedy prvek smažeme a později zase vložíme, je to jako kdyby se vkládal úplně nový prvek.

Příklad

Pro lepší pochopení fungování algoritmu TS(p) uvedeme jednoduchý příklad. Nejprve vložíme pět prvků a pak provedeme několik vyhledání. U jednotlivých operací se buď provádí větev a) algoritmu (MTF), nebo větev b) (TS).

1. vkládání - pokud se použije MTF, tak vložený prvek bude přesunut na začátek, pokud se použije TS, tak zůstane na konci, protože tím, že to je první požadavek na daný prvek, tak se při výběru strategie TS kromě zaznamenání požadavku nic nedělá (pokud by se vyskytlo opakované vložení nějakého prvku, algoritmus by se choval jako u úspěšného vyhledání – viz další část). V následujícím popisu je vždy uvedena operace s argumentem, použitá větev algoritmu

(MTF nebo TS) a podoba spojového seznamu po dané operaci (začátek seznamu je vlevo).

$\texttt{Insert}(15), \texttt{MTF} : 15$

$\texttt{Insert}(10), \texttt{TS} \ : 15 - 10$

$\texttt{Insert}(30), \texttt{TS} \ : 15 - 10 - 30$

$\texttt{Insert}(45), \texttt{MTF} : 45 - 15 - 10 - 30$

$\texttt{Insert}(25), \texttt{TS} \ : 45 - 15 - 10 - 30 - 25$

2. vyhledání - při použití MTF je situace opět jednoduchá, prvek je přesunut na začátek seznamu, při použití TS je situace složitější, proto provedené přesuny zdůvodníme u každé takové operace zvlášť (zmiňovaný $v_x(t)$ je definován v popisu algoritmu).

$\texttt{Member}(10), \texttt{MTF} : 10 - 45 - 15 - 30 - 25$

$\texttt{Member}(15), \texttt{TS} \ : 10 - 45 - 15 - 30 - 25$

Od posledního požadavku na 15 bylo u 10 i 45 použito MTF, takže 15 nikam dopředu přesunuta nebyla ($v_{15}(7) = 15$).

$\texttt{Member}(25), \texttt{TS} \ : 10 - 25 - 45 - 15 - 30$

Od posledního požadavku na 25 bylo použito u 10 MTF a u 15 jednou TS, takže 25 byla přesunuta za 10 ($v_{25}(8) = 45$).

$\texttt{Member}(30), \texttt{MTF} : 30 - 10 - 25 - 45 - 15$

$\texttt{Member}(25), \texttt{TS} \ : 30 - 25 - 10 - 45 - 15$

Od posledního požadavku na 25 bylo použito u 30 MTF a u ostatních prvků nic, takže 25 byla přesunuta za 30 ($v_{25}(10) = 10$).

$\texttt{Member}(15), \texttt{TS} \ : 30 - 25 - 15 - 10 - 45$

Od posledního požadavku na 15 bylo u 30 použito MTF a na 25 byly dva požadavky, takže 15 bude přesunuta za tyto dva prvky ($v_{15}(11) = 10$).

Hledání $v_x(t)$

Nechť argumentem operace v čase t je x. Podmínku „předchází x a (i) … nebo (ii) … “ z popisu algorimtu označíme jako $C(x,t)$. Tedy $v_x(t)$ je prvek nejblíže začátku seznamu splňující $C(x,t)$. Tento prvek nelze určit při prvním průchodu, kdy se hledá x, protože podmínka $C(x,t)$ závisí na x (na „historii“ požadavků na něj). Pro nalezení $v_x(t)$ je potřeba další průchod seznamem, který budeme někdy v dalším textu pro zkrácení nazývat „druhý průchod u TS(p)“. Protože hledáme prvek nejblíže začátku seznamu, zdálo by se, že je potřeba seznam procházet odpředu, ale ukážeme, že je možné jít odzadu. Myšlenka bude následující – pokud nějaký prvek y splňuje $C(x,t)$, tak i všechny další prvky, které jsou v seznamu mezi y a x splňují $C(x,t)$.

Následující lemma dokázala Suzanne Albers [1].

Lemma 3.1. Nechť t je čas z $[1,m]$ a x prvek požadovaný v čase t. Pokud byl x požadován alespoň jednou v $[1,t-1]$, pak platí následující dvě tvrzení. Nechť t', $t' \leq t$, označuje čas posledního dřívějšího (do času $t-1$ včetně) požadavku na x.
a) Pokud prvek y, $y \neq x$, byl požadován aspoň dvakrát během $[t', t-1]$, pak y předchází $v_x(t)$ v seznamu v čase t.
b) Pokud prvek y, $y \neq x$, byl požadován právě jednou během $[t', t-1]$ a odpovídající požadavek byl obsloužen krokem (a) algoritmu TS(p), pak y předchází $v_x(t)$ v seznamu v čase t.

Důkaz. Předpokládejme, že existuje čas v $[1,m]$, ve kterém lemma neplatí. Pak nechť $t_0 \in [1,m]$ je nejdřívější čas, ve kterém je lemma porušeno, a x prvek požadovaný v čase t_0. Dále nechť t'_0, $t'_0 < t_0$, je čas, ve kterém prvek $x = s(t_0)$ byl požadován naposledy před časem t_0, a nechť $z = v_x(t_0)$. Existence t'_0 plyne z porušenosti lemmatu v čase t_0.

Nejprve vyšetříme případ, že neplatí tvrzení a) lemmatu. V tomto případě existuje prvek y, $y \neq x$, který je požadován alespoň dvakrát v $[t'_0, t_0-1]$ a nepředchází $v_x(t_0)$ v čase t_0. Nechť t_y je čas posledního požadavku na y v $[t'_0, t'_0-1]$. Ukážeme, že po vyřízení požadavku v čase t_y, prvek y předchází v seznamu prvek z. Pokud $s(t_y)$ je obsloužen pomocí kroku (a) algoritmu, pak je to vidět hned. Předpokládejme, že $s(t_y)$ je obsloužen pomocí kroku (b) algoritmu. Podle definice $v_x(t_0)$, prvek z je požadován nejvýše jednou v $[t'_0, t_0-1]$ a tento požadavek je obsloužen pomocí kroku (b) algoritmu. Z toho plyne, že když TS(p) vyřizuje požadavek $s(t_y)$, z nemůže předcházet $v_y(t_y)$ (z totiž (i) nebo (ii) z kroku (b) algoritmu splňuje, protože je to $v_x(t_0)$). Tedy y je vloženo (nebo spíše přesunuto) na nějakou pozici před z. Tím jsme tedy ukázali, že y musí předcházet z po obsloužení $s(t_y)$. Protože (podle předpokladu) z předchází y v čase t_0, tak prvek z musí být požadován v nějakém čase $t_z \in [t_y+1, t_0-1]$ a z musí být vložen na nějakou pozici před y při vyřízení $s(t_z)$ (tedy se mj. nebude jednat o první požadavek na z). Z toho plyne, že y nemůže předcházet $v_z(t_z)$ v čase t_z, protože $s(t_z)$ je obslouženo pomocí kroku (b) algoritmu. Zároveň v čase t_z platí, že y byl požadován aspoň dvakrát od posledního požadavku na z (protože v $[t'_0, t_0-1]$ nejvýše jeden požadavek na z). Tím dostáváme spor s minimalitou t_0.
Nyní předpokládejme, že je porušeno tvrzení (b) lemmatu. Nechť y, $y \neq x$, je prvek, pro který tvrzení (b) lemmatu neplatí, a nechť $t_y \in [t_y+1, t_0-1]$ je čas požadavku na y. Dle předpokladu y nepředchází $z = v_x(t_0)$ v čase t_0. Protože $s(t_y)$ je obslouženo pomocí kroku (a) algoritmu, y předchází z po vyřízení $s(t_y)$. Dále už pokračujeme jako v předchozím případě a opět tedy vyvodíme spor s minimalitou t_0. Q.E.D.

Z lemmatu tedy plyne, že pokud prvek y leží v čase t mezi x a $v_x(t)$, tak splňuje podmínku $C(x,t)$, protože pokud by nesplňoval (i) ani (ii) z kroku (b), tak by podle lemmatu musel ležet před $v_x(t)$. Prvky splňující $C(x,t)$ tvoří vlastně souvislý pás od $v_x(t)$ až po bezprostředního předchůdce prvku x. Tedy $v_x(t)$ je

opravdu možné hledat i odzadu, a to tak, že půjdeme od prvku x (nebo tedy spíše od jeho bezprostředního předchůdce) a procházíme řetězec směrem k začátku, dokud nenarazíme na první prvek, který $C(x,t)$ nesplňuje, nebo na začátek řetězce. V prvním případě přesuneme x za daný prvek nesplňující $C(x,t)$, v druhém případě na začátek řetězce. (V obou případech platí, že prvek x je vložen bezprostředně před $v_x(t)$.)
Otázkou je, jestli je výhodnější postupovat zepředu nebo zezadu. K tomuto problému se vrátíme v kapitole 3.4.

Okrajové případy hodnot p

Pro $p = 0$ a $p = 1$ je TS(p) deterministický algoritmus. V případě $p = 1$ je to přesně strategie MTF, v případě $p = 0$ je to algoritmus označovaný jako TS.

3.2 Strategie Sorted

Tato strategie spočívá pouze v tom, že se řetězce udržují setříděné podle hodnot klíčů (lze tedy používat, pokud jsou hodnoty klíčů navzájem porovnatelné). Tedy na rozdíl od předchozích strategií zde to nebude fungovat tak, že by se udělala „klasická“ operace a pak nějaký přesun prvků, ale setříděnost má ten efekt, že při procházení řetězce není nutné dojít až na konec řetězce, abychom věděli, že tam prvek není. Pokud jsou hodnoty v řetězci setříděné například vzestupně, stačí dojít k prvku s klíčem větším, než je klíč v argumentu operace. V případě úspěšného Insert je tedy prvek vložen před první nalezený prvek s větším klíčem, příp. na konec řetězce. V případě neúspěšného Member a Delete také není potřeba procházet řetězec dál než do prvního většího klíče (v případě vzestupného setřídění). Případy, kdy argument operace v řetězci je, tedy neúspěšný Insert a úspěšný Member a Delete, jsou vlastně stejné jako u klasické strategie – při průchodu řetězcem se dojde k danému prvku a v případě Delete se tento prvek smaže (u zbylých dvou operací se už nedělá nic).

3.2.1 Složitost

V následujícím odstavci se zmíníme o složitosti hašovaní s použitím strategie Sorted. Opět budeme jako měřítko složitosti brát počet testů a budeme složitost odhadovat za stejných předpokladů jako v kapitole 2. V případě úspěšného vyhledání je očekávaný počet testů stejný jako u klasické strategie, lze ho tedy odhadnout výrazem $1 + \frac{\alpha}{2}$. Při neúspěšném vyhledání je očekávaný počet testů logicky nižší než u klasické strategie, konkrétně ho lze aproximovat výrazem $e^{-\alpha} + 1 \ + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$. Uváděné odhady lze najít např. v [22].

3.3 Strategie BVS - binární vyhledávací strom

V této strategii je řetězec reprezentován jako binární vyhledávací strom. Není u něj použito žádné vyvažování, ani žádné přesuny prvků po provedení „klasických“ operací podobné těm u MTF nebo TR.

Binární vyhledávací strom je strom s kořenem, ve kterém každý uzel má žádného, jednoho nebo dva syny. Každý uzel obsahuje prvek (klíč) a platí, že prvek v uzlu u je větší než prvky uložené v uzlech v levém podstromu uzlu u (tj. levý syn plus jeho potomci) a menší než prvky uložené v pravém podstromu uzlu u.

Tak jako v případě seznamu byl průchod seznamem složen z jednotlivých přesunů na další prvek, tak v případě stromu je průchod složen z přesunů z uzlu na jednoho z jeho synů. Pokud je argument operace menší než klíč v uzlu, přesuneme se na levého syna, pokud je větší, přesuneme se na pravého syna. Průchod stromem končí, pokud narazíme na argument operace nebo na konec cesty (ukazatel NIL). Tento průchod stromem je součástí operací Member, Insert i Delete.

3.3.1 Operace

Member

Při průchodu stromem buď hledaný prvek najdeme, nebo narazíme na ukazatel NIL, což znamená, že hledaný prvek v celém stromě není.

Insert

Pokud při průchodu stromem nalezneme prvek, který se má vkládat, skončíme. V opačném případě, pokud dojdeme při průchodu na ukazatel NIL, tak nový prvek vložíme tak, že vytvoříme nový uzel, na který bude ukazovat právě tento ukazatel.

Delete

Předpokládejme, že chceme mazat uzel u s klíčem x.

Nejprve vezmeme případ, kdy u nemá žádného syna. Pokud u je kořen, tak ho smažeme a ukazatel na kořen stromu nastavíme na NIL. Pokud u není kořen, tak označme v jeho otce. Prvek u smažeme a ukazatel ve v, který ukazoval na u, nastavíme na NIL.

Nyní se podíváme na případ, kdy u má jen jednoho syna, nechť je to pravý syn. Pokud u je kořen, tak novým kořenem se stane pravý syn uzlu u. Pokud u není kořen a je pravým synem svého otce v, tak ho smažeme a novým pravým synem uzlu v bude pravý syn uzlu u. (Obdobně pokud u bude levým synem v, tak novým levým synem v se stane pravý syn uzlu u.) Případ, kdy u má jen levého syna, se udělá analogicky.

Komplikovanější případ je, pokud uzel u má oba syny. Postupujeme tak, že nalezneme nejpravějšího potomka uzlu u z jeho levého podstromu – označme ho w. Klíč z w překopírujeme do u a odstraníme w (w má nanejvýš jednoho syna, takže to lze provést pomocí postupu z předešlého odstavce). Tím jsme se zbavili klíče x a díky tomu, že w byl nejpravější potomek z levého podstromu, tak i nově vzniklý strom je korektní BVS.

3.3.2 Složitost

Zde uvedeme některé teoretické odhady týkající se náhodných binárních vyhledávacích stromů, které jsou převzaty z [19] a [16]. Náhodné BVS se dají definovat jako BVS vzniklé postupným vkládáním náhodné permutace klíčů do původně prázdného stromu. Podobně také mohou vzniknout vkládáním klíčů vygenerovaných na základě rovnoměrného rozdělení, což je způsob, jakým budou vznikat v našich experimentech.

Očekávaná výška

Očekávaná výška náhodného BVS s n prvky je $c \ln(n) - d \ln\ln(n) + O(1)$, kde $c \approx 4.311$ a $d \approx 1.953$.

Počet porovnání při konstrukci BVS

Průměrný počet porovnání vykonaných při konstrukci BVS vložením n různých prvků v náhodném pořadí do původně prázdného stromu je $2(n+1)(H_{n+1}-1)-2n$ (H_n značí n-té harmonické číslo a dá se aproximovat jako $\ln(n) + 0.57721$). Počet testů je o jedničku větší než počet porovnání (to je způsobeno rozdílem u vkládání prvního prvku).

Očekávaný počet porovnání při úspěšném a neúspěšném vyhledání v BVS

V náhodném BVS s n prvky je za předpokladu stejné pravděpodobnosti vyhledávání všech prvků z univerza očekávaný počet porovnání při úspěšném vyhledání $2H_n - 3 - 2H_n/n$ a očekávaný počet porovnání při neúspěšném vyhledání $2H_{n+1} - 2$. Počet testů je v tomto případě roven počtu porovnání (protože předpokládáme $n > 0$).

3.4 Implementace seznamu

Při operacích na seznamech potřebujeme často po nalezení prvku přistupovat k jeho předchůdci, příp. několika předchůdcům. Pokud je seznam implementován jako jednosměrný seznam, potom je nutné si při procházení seznamu udržovat ukazatele i na předchůdce prvků, které testujeme. Proto se nabízela otázka, jestli by nebylo lepší použít obousměrné seznamy, kde se dají ukazatele na předchůdce jednoduše dostat až v momentě, kdy je opravdu potřebujeme. Navíc pokud bychom chtěli u strategie TS(p) aplikovat druhý průchod zezadu, tak by použití obousměrné implementace u této strategie bylo nutností. Použití obousměrného seznamu namísto jednosměrného přináší samozřejmě nevýhodu v podobě nutnosti aktualizování více ukazatelů při úpravách seznamu. Abychom zjistili, která verze je efektivnější, provedli jsme experimenty, ve kterých jsme pro jednotlivé strategie vyzkoušeli rychlost základních operací u obou verzí. Rychlost jsme porovnávali na základě naměřených časů a testy jsme prováděli pro rovnoměrné rozdělení, a to jak pro hašovací tabulku, tak pro samotný seznam. Grafická podoba výsledků testů je na přiloženém DVD, zde výsledky pouze slovně zhodnotíme.

U nesamoupravujících strategií jsme testovali operace Insert a Delete. Operaci Member jsme testovat nemuseli, protože u té se prováděné operace u obou verzí vůbec neliší. Naměřené časy byly u obou verzí dost podobné, jako rychlejší se jevila ale přecejen spíše jednosměrná implementace, a tak byla upřednostněna ta.

U samoupravujících strategií jsme testovali operace Insert, úspěšný Member a neúspěšný Member. Operace Delete se už testovat nemusí, protože ta je stejná jako u klasické strategie, kde už jsme ji otestovali (viz výše).

U strategie TS(p) jsme u obousměrné verze testovali dvě varianty, tu s druhým průchodem zepředu a tu s druhým průchodem zezadu, takže jsme vlastně porovnávali tři verze implementace.

Výsledky byly takové, že hovořily ve prospěch obousměrné verze (u TS(p) ve prospěch varianty s druhým průchodem zezadu) nebo byly v podstatě vyrovnané. Operace Delete testovaná už u klasické verze se zdála rychlejší u jednosměrné verze, ale opravdu zanedbatelně, zatímco operace Member byla u jednosměrné verze mnohdy výrazně pomalejší. Proto jsme se rozhodli použít u MTF a TR obousměrnou verzi, u TS(p) obousměrnou verzi s druhým průchodem zezadu.

Kapitola 4

Samoupravující strategie pro nesetříděné spojové seznamy

4.1 Způsoby porovnávání algoritmů

Motivací pro výzkum v oblasti list-update problému (problému udržování seznamu nesetříděných prvků) je podobně jako jinde nalezení nějakého lepšího algoritmu. Otázkou ale zůstává, kdy se dá říci, že nějaký algoritmus je lepší. V této kapitole jsou ukázány nejpoužívanější způsoby porovnávání algoritmů pro list-update problém a u každého z nich dostupné výsledky týkající se algoritmů testovaných v této práci, případně i dalších.

4.1.1 Kompetitivita

Jednou z možností určení, jak je algoritmus dobrý, je využít amortizovanou složitost a zkoumat, kolikrát může být daný algoritmus pro sekvenci požadavků pomalejší než optimální postup. Aby to bylo možné porovnávat, je potřeba zavést jakési ceny za zpracování požadavků, zavést jakýsi platební model (cost model). Ten standardní vypadá následovně. Cena za úspěšnou operaci Member nebo Delete je i, kde i je pozice argumentu operace. Cena úspěšné operace Insert je $n+1$, kde n je počet prvků v seznamu před operací.

Po operaci Member nebo Insert může být argument operace přesunut za nulovou cenu blíže k začátku seznamu. Výměny prováděné při tomto posunu, se nazývají volné. Ostatní výměny jsou placené a je u nich za prohození sousedních prvků v seznamu stanovena cena 1. Cena za sérii požadavků pak zahrnuje součet cen za všechny provedené operace Member, Delete, Insert a za všechny placené výměny.

V této práci se zabýváme on-line algoritmy, tedy takovými, které při zpracovávání požadavků neznají budoucí požadavky. On-line algoritmy se často porovnávají s optimálním off-line algoritmem, což je algoritmus, který zná dopředu celou sérii požadavků a umí ji zpracovat za optimální cenu. Na základě tohoto srovnání je zaveden pojem c-kompetitivity.

Definice 4.1. Mějme deterministický on-line algoritmus A. Označme s sérii požadavků, $C_A(s)$ cenu, za kterou s zpracuje algoritmus A, a $C_{OPT}(s)$ cenu, za kterou s zpracuje optimální off-line algoritmus OPT. Potom algoritmus A nazveme c-kompetitivní, pokud existuje konstanta a taková, že pro všechny série požadavků s a seznamy všech délek platí

$$C_A(s) \leq c \ \cdot C_{OPT}(s) + a.$$

Toto c se nazývá kompetitivní poměr.

Kompetitivita u randomizovaných on-line algoritmů je definována s ohledem na protivníka. Existují tři druhy protivníků (viz [7]), zapomínající, adaptivní online a adaptivní off-line, ale nás bude zajímat pouze kompetitivita vůči zapomínajícímu protivníkovi. Zapomínající protivník musí generovat celou sérii požadavků dopředu, dříve než jsou jakékoliv požadavky on-line algoritmem zpracovávány, a je mu účtována za zpracování sekvence požadavků cena optimálního off-line algoritmu.

Definice 4.2. Mějme randomizovaný on-line algoritmus A. Označme s sérii požadavků, $C_A(s)$ cenu, za kterou s zpracuje algoritmus A, a $C_{OPT}(s)$ cenu, za kterou s zpracuje optimální off-line algoritmus OPT. Potom algoritmus A nazveme c-kompetitivní vůči zapomínajícímu protivníkovi, pokud existuje konstanta a taková, že pro všechny série požadavků s a seznamy všech délek platí

$$E\left[C_A(s)\right] \leq c \ \cdot C_{OPT}(s) + a,$$

kde očekávaná hodnota je brána přes náhodné volby algoritmu A.

Dále budeme u randomizovaných algoritmů používat pojem „c-kompetitivita" ve smyslu „c-kompetitivita vůči zapomínajícímu protivníkovi".

Prvními, kdo se výrazněji zabývali kompetitivitou, byli Sleator a Tarjan [20], kteří dokázali, že algoritmus MTF je 2-kompetitivní, zatímco algoritmus TR není c-kompetitivní pro žádnou konstantu c.

Pokud navíc budeme uvažovat seznam konkrétní délky l, pak kompetitivní poměr pro MTF je $2 - 2/(l+1)$ (viz [12]) a pro TR $2l/3$ (viz [8]).

Karp a Raghavan [13] ukázali, že tím, že je MTF 2-kompetitivní, je už mezi deterministickými algoritmy optimální.

Věta 4.3. Nechť A je deterministický on-line algoritmus (pro list-update problém). Pokud A je c-kompetitivní, pak $c \geq 2$.

Pro konkrétní délku seznamu l je spodní hranice (pro deterministické algoritmy) ještě nižší než 2, konkrétně $2 - 2/(l+1)$ (viz [13]), což přesně odpovídá kompetitivnímu poměru u MTF pro tento případ, a tedy i pro konkrétní délku seznamu je MTF (mezi deterministickými algoritmy) optimální.

Kompetitivita je založena na porovnání s optimálním (dynamickým) off-line algoritmem, existují však i odhady vůči optimálnímu statickému off-line algoritmu. Optimální statický off-line algoritmus (někdy je užíván název optimální statické pořadí – optimal static ordering (OSO)) nejprve seřadí prvky v seznamu podle jejich pravděpodobností neklesajícím způsobem a pak už pouze vyřizuje požadavky bez toho, aby nějak měnil pořadí prvků. Bentley a McGeoch [8] například ukázali, že

$$C_{MTF}(s) \leq 2 \ \cdot C_{OSO}(s) + a$$

a že neexistují konstanty c, b, aby

$$C_{TR}(s) \leq c \ \cdot C_{OSO}(s) + b.$$

Navíc u MTF dokázali, že faktor 2 už není možné zlepšit. Přestože tedy OSO je slabší než OPT, tak výsledky u MTF a TR jsou pro statický i dynamický off-line algoritmus stejné.

U deterministických algoritmů je nejlepší možný kompetitivní poměr 2, u randomizovaných je dosahováno výrazně lepších výsledků. Jedním z takových algoritmů je i námi testovaný TS(p) zahrnující díky volbě velikosti p celou (nekonečnou) rodinu algoritmů.

Věta 4.4. Pro každé reálné číslo $p \in [0, 1]$ je algoritmus TS(p) c-kompetitivní, kde $c = max\,\{2 - p, 1 + p(2 - p)\}$.

Důkaz. [1]

Nejnižšího kompetitivního poměru dosahuje TS(p) pro $p = (3 - \sqrt{5})/2$, kdy dostáváme q-kompetitivní algoritmus, kde $q = (1 + \sqrt{5})/2 \approx 1.62$. Pokud je p rovno 0 nebo 1, dává TS(p) deterministický algoritmus. V případě $p = 1$ se jedná o MTF algoritmus, v případě $p = 0$ o algoritmus označováný jako TS(0) nebo někdy také jenom TS. Jak plyne z uvedené věty, algoritmus TS(0) je stejně jako MTF 2-kompetitivní.

Nejlepším algoritmem je v současnosti algoritmus Combination, který je založen na kombinaci algoritmů Bit a TS(0). Algoritmus Bit funguje tak, že každý prvek má u sebe bit, který se při každém požadavku na prvek převrací. Na začátku jsou tyto bity nastavovány náhodně na 0 a 1 s rovnoměrným rozdělením a pak se při požadavku na prvek uplatňuje pravidlo move-to-front (přesun prvků na začátek seznamu), jen pokud se bit převrátil na jedničku. Algoritmus Combination pracuje tak, že s pravděpodobností 0.8 použije strategii Bit a s pravděpodobností 0.2 strategii TS(0).

Věta 4.5. Algoritmus Combination je 1.6-kompetitivní.

Důkaz. [4]

Díky algoritmu Combination tedy víme, že je možné dosáhnout poměru 1.6. Tím je daný jakýsi horní odhad pro optimální algoritmus. B. Teia [21] dokázal naopak (do této doby) nejlepší dolní odhad.

Věta 4.6. Nechť A je randomizovaný on-line algoritmus pro list-update problém. Pokud A je c-kompetitivní, pak $c \geq 1.5$.

Z hlediska kompetitivity je tedy z testovaných samoupravujících strategií nejlepší TS(p), pak MTF (ta je na stejné úrovni s TS(0)) a jako nejhorší vychází TR.

Projektivita

Většina známých algoritmů pro list-update problém vykazuje vlastnost projektivity, což znamená, že vzájemné pořadí dvou prvků v seznamu závisí pouze na požadavcích na tyto prvky a počátečním stavu seznamu (příkladem neprojektivního algoritmu je třeba TR). Projektivita je velice výhodná při teoretické analýze, protože umožňuje se omezit na seznamy se dvěma prvky. Pro projektivní algoritmy bylo dokázáno, že nemohou dosáhnout lepšího kompetitivního poměru než 1.6 (viz [5]). To znamená, že výše zmíněný algoritmus COMB je optimální v rámci projektivních algoritmů, a že pokud existuje nějaký lepší algoritmus, tak musí být neprojektivní.

4.1.2 Očekávaná složitost

Další možností pro určení náročnosti algoritmu je využít očekávanou složitost. Pro tyto účely budeme uvažovat pouze sekvence požadavků generované dle nějakého pravděpodobnostního rozdělení a navíc budeme předpokládat pouze operaci úspěšného vyhledání. V seznamu nechť jsou prvky $x_1, ..., x_n$ a $\mathbf{p} = (p_1, ..., p_n)$ je pravděpodobnostní vektor, kde p_i je pravděpodobnost vyhledávání prvků x_i. Veličinou, kterou budeme porovnávat u jednotlivých algoritmů, bude asymptotická očekávaná cena. Pro algoritmus A označíme $E_A(\mathbf{p})$ asymptotickou očekávanou cenu pro algoritmus A za jeden požadavek v sérii požadavků vygenerované na základě pravděpodobnostního rozdělení $\mathbf{p}$. Formální definice je např. v [2]. Asymptotická očekávaná cena on-line algoritmu bývá srovnávána s asymptotickou očekávanou cenou optimálního off-line algoritmu nebo s asymptotickou očekávanou cenou optimálního statického off-line algoritmu (OSO). Pro asymptotickou očekávanou cenu algoritmu OSO platí

$$E_{OSO}(\mathbf{p}) = \sum_{i=1}^{n} i \cdot p_i.$$

Asymptotická očekávaná cena pro algoritmus MTF byla odvozena např. v [18]:

$$E_{MTF}(\mathbf{p}) = 1 + 2 \sum_{1 \leq i \ < j \ \leq n} \frac{p_i p_j}{p_i + p_j}$$

Co se týče srovnání asymptotické očekávané ceny pro algortimus MTF a OSO, tak Chung s kolegy [10] ukázal, že platí

$$E_{MTF}(\mathbf{p}) \leq \frac{\pi}{2} E_{OSO}(p).$$

Obecně už nelze tento poměr zlepšit (viz [11]), ale existují pravděpodobnostní rozdělení $\mathbf{p}$, u kterých poměr $E_{MTF}(\mathbf{p})/E_{OSO}(\mathbf{p})$ je nižší než $\pi/2 \approx 1.5708$. Mezi takováto rozdělení patří například Zipfův zákon nebo geometrické rozdělení, u kterých $E_{MTF}(\mathbf{p})/E_{OSO}(\mathbf{p}) \leq 2 \ln 2 \approx 1.3863$.

Pro TR žádný výraz (v podobě uzavřené formule) vyjadřující asymptotickou očekávanou cenu odvozen nebyl, ale Rivest [18] ukázal, že asymptotická očekávaná cena u TR je nižší než u MTF.

Věta 4.7. Pro každé pravděpodobnostní rozdělení $\mathbf{p} = (p_1, ..., p_n)$ je $E_{TR}(\mathbf{p}) \leq E_{MTF}(\mathbf{p})$. Nerovnost je ostrá kromě případů, kdy $n = 2$ nebo $p_i = 1/n$ pro $i = 1, ..., n$.

V tomto hledisku tedy vychází TR jako lepší, na druhou stranu Bitner [9] ukázal, že sice TR má nižší asymptotickou očekávanou cenu než MTF, ale MTF k asymptotické očekávané ceně konverguje rychleji než TR. Gonnet [11] dokázal, že TR potřebuje $\Theta(n^2)$ vyhledání, aby se dostala pod hranici $(1+\epsilon)\cdot$(asymptotická cena).

Albers a Mitzmacher [2] ukázali, že TS(0) má pro všechna nerovnoměrná rozdělení nižší asymptotickou očekávanou cenu než MTF.

Věta 4.8. Pro každé pravděpodobnostní rozdělení $\mathbf{p}$ platí

$$E_{MTF}(\mathbf{p}) - E_{TS(0)}(\mathbf{p}) = \sum_{1 \leq i \ \leq j \ \leq n} p_i p_j \frac{(p_i - p_j)^2}{(p_i + p_j)^3}.$$

Dále také získali odhady týkající se porovnání algoritmu TS(0) s OSO a s optimálním off-line algoritmem.

Věta 4.9. Pro každé pravděpodobnostní rozdělení $\mathbf{p}$ platí $E_{TS(0)}(\mathbf{p}) \leq 1.34\, E_{OSO}(\mathbf{p})$.

Věta 4.10. Pro každé pravděpodobnostní rozdělení $\mathbf{p}$ platí $E_{TS(0)}(\mathbf{p}) \leq 1.5\, E_{OPT}(\mathbf{p})$, kde OPT je optimální off-line algoritmus.

Algoritmus TS(0) je vlastně jediným algoritmem, u něhož známe, co se týče asymptotické očekávané ceny, také srovnání s optimálním (dynamickým) off-line algoritmem, u ostatních algoritmů existují nanejvýše srovnání s optimálním statickým off-line algoritmem.

Z hlediska očekávané složitosti je tedy z testovaných samoupravujících strategií (se známým odhadem asymptotické očekávané ceny) nejhorší MTF. TS(0) a TR jsou lepší než MTF a jejich vzájemné srovnání asi není známo. Jaké jsou výsledky pro TS(p), kde $p \neq 0, 1$, se nám nepodařilo zjistit.

4.2 Dřívější experimenty

V této části bych chtěl zmínit některé experimenty z minulosti zabývající se samoupravujícími strategiemi. Soustředím se hlavně na výsledky týkající se strategií zkoumaných v této práci.

Rivest [18] prováděl experiment, kdy měl prvky uložené v seznamu a při vyhledávání použil sekvence požadavků, které odpovídaly Zipfovu zákonu. Pro tyto sekvence vykazoval lepší výsledky TR než MTF.

Bentley a McGeoch [8] použili také pouze samotný seznam a sekvence požadavků odpovídající sekvenci slov v textových souborech a zdrojových souborech jazyka Pascal (klíče jsou tedy slova). V jejich testech měla strategie TR na rozdíl od Rivestova experimentu horší výsledky než MTF. Obě tyto strategie byly také pomalejší než OSO s výjimkou dvou zdrojových souborů, kde byl MTF dokonce nepatrně lepší než OSO.

Zobel, Heinz a Williams [24] použili podobný postup jako Bentley a McGeoch, ale místo samotného seznamu použili hašovací tabulku s metodou separovaných řetězců a namísto malých souborů velké soubory anglických slov. Distribuce by tedy měla přibližně odpovídat Zipfovu zákonu. Testovali strategii MTF oproti klasické strategii a v experimentech jim vycházela MTF pro tabulky s vysokým faktorem naplnění (v řádu stovek) až dvakrát rychlejší, pro tabulky s nižším faktorem naplnění už nebyl rozdíl takový, ale stále byla MTF znatelně rychlejší. Ještě bych poznamenal, že tento experiment je jediným nalezeným otestováním samoupravujících strategií v hašovacích tabulkách v praxi.

Bachrach a El-Yaniv [6] prováděli rozsáhlý experiment, kdy měli samotný seznam a použili pro generování sekvence požadavků soubory z velkého korpusu a porovnávali mnoho samoupravujících strategií. V jejich experimentech překvapivě deterministické algoritmy předčily ty randomizované. Podobně jako v [8] byla strategie MTF opět rychlejší než strategie TR.

Kapitola 5

Testování

5.1 Použitá pravděpodobnostní rozdělení dat

5.1.1 Rovnoměrné rozdělení

Toto rozdělení není potřeba dlouze představovat, jeho myšlenka je velmi jednoduchá – každé číslo z daného intervalu má stejnou pravděpodobnost. Má dvě verze, diskrétní a spojitou, a obě byly při našich experimentech použity.

Diskrétní verze

Pravděpodobnostní funkce rovnoměrného diskrétního rozdělení je

$$P(X = x) = 1/K, x = 0, 1, ..., K - 1$$

Pokud se budeme chtít omezit na L čísel, kde $L < K$, můžeme použít funkci modulo L. Abychom takto dostali opět rovnoměrné rozdělení, tak je potřeba, aby $L|K$. Na druhou stranu je pravda, že i když L nedělí K, tak pokud je L o několik řádů menší, tak „chyba“ vůči rovnoměrnosti bude velmi malá a bude to skoro rovnoměrné rozdělení. Tu vhodnost dělitelnosti jsme si uvědomili až po určitém čase, a tak u souboru s klíči je ještě použito $K = 2^{32}$ (a přitom testované velikosti tabulek nejsou mocniny dvojky). Asi to ale nevadí, protože pořád se to dá považovat za případ, kdy hašovaci funkce (v našem případě modulo velikost tabulky) rozděluje univerzum klíčů rovnoměrně. U klíčů to navíc vadí ještě méně, než kdyby to bylo u indexů (indexy do pole vložených prvků – viz 5.2.3), protože velikosti tabulek byly v testech maximálně 100 tisíc, zatímco počty prvků až 5 milionů, takže tam by ta „chyba“ byla větší. Při generování indexů už bylo použito $K = 2520 \cdot 10^6$, u kterého právě platí, že je dělitelné všemi použitými počty prvků (500 tis., 1 mil., 2 mil. apod.)

Spojitá verze

Hustota rovnoměrného spojitého rozdělení je

$$f(x) = \begin{cases} 1/|I| & \text{pro } x \in I \\ 0 & \text{pro } x \notin I, \end{cases}$$

kde I je interval z R

Zde by omezování na určitý podinterval fungovalo podobně jako u diskrétní verze, ale při experimentech to ani nebylo nutné.

5.1.2 Geometrické rozdělení

Rovnoměrné rozdělení je jednoduché na představu, ale v praxi často dochází k tomu, že rovnoměrnost moc neplatí. To bylo motivací k tomu použít při testování kromě rovnoměrného i nějaké nerovnoměrné rozdělení (zde máme na mysli rozdělení diskrétní, spojité bylo potřeba pouze k jednomu účelu, kde jsme naopak potřebovali rovnoměrné). Rozhodli jsme se použít rozdělení geometrické. Jeho pravděpodobnostní funkce je

$$P(x) = p(1-p)^x, p \ \in (0,1), x \ = 0,1,2,\dots$$

Někde se uvádí verze od jedničky (tedy $x = 1, 2, \dots$), ale v našem případě byla vhodnější verze od nuly. Parametr p se nazývá koeficient geometrického rozdělení. Byla použita rozdělení s koeficienty 0.001, 0.0001, 0.00001, 0.000001 (více k tomu v kapitole 5.2).

Podobně jako u rovnoměrného rozdělení, i zde se budeme chtít omezit na určitý rozsah $[0, L-1]$. Geometrické rozdělení má tu vlastnost, že po uplatnění operace modulo L (zde je důležité, že je to verze „od nuly", kdyby to byla verze „od jedničky", tak by se to omezení muselo dělat trochu jinak – odečíst 1, udělat modulo L, přičíst 1) na nagenerovaná čísla, budou mít získaná čísla zase jakoby „geometrické" rozdělení, bude to vlastně to původní rozdělení přenásobené určitým koeficientem:

Po uplatnění operace modulo L máme pro $x \in [0, L-1]$ pravděpodobnostní funkci

$$P'(x) = \sum_{y=x+kL, k \ \in N} p(1-p)^y = p \sum_{y=x+kL, k \ \in N} (1-p)^y =$$

$$= p\frac{(1-p)^x}{1-(1-p)^L} = \frac{1}{1-(1-p)^L} P(x)$$

Tedy nová pravděpodobnostní funkce je vlastně ta původní přenásobená koeficientem $\frac{1}{1-(1-p)^L}$. (To mj. znamená, že $P'(x+1)/P'(x) = P(x+1)/P(x) = 1-p$.) Jak plyne z její definice, je splněno $\sum_{x=0}^{L-1} P'(x) = 1$ (což je nutná podmínka pro korektní definici).

Jiný způsob, jak se omezit na interval $[0, L-1]$, by bylo příliš vysoké nagenerované hodnoty „zahazovat". Pravděpodobnosti čísel v intervalu $[0, L-1]$ by se tím zvýšily, přenásobily by se vlastně určitým koeficientem tak, aby dávaly v součtu 1. To znamená, že takto bychom dostali to samé rozdělení jako při použití modulu.

V následujícím textu budeme již uvažovat to omezené geometrické rozdělení na čísla $0, \dots, L-1$. Při testování jsme používali tři verze geometrického rozdělení.

Základní verze

To je klasická verze geometrického rozdělení, tak jak ho generují obvykle generátory (použije se tedy pouze modulo L k omezení na interval $[0, L-1]$).

Inverzní verze

Tuto verzi jsme nazvali inverzní, protože platí, že pravděpodobnost prvku x z $[0, L-1]$ bude rovna pravděpodobnosti prvku $L-1-x$ v klasické verzi. Implementace tohoto vypadala tak, že se nagenerované číslo nahradilo svým „protějškem“ v rámci intervalu $[0, L-1]$.

Náhodná verze

Tuto verzi jsme označili jako náhodnou, protože pravděpodobnosti jednotlivých prvků z intervalu získáme zpermutováním jejich pravděpodobností z klasické verze. V praxi se to udělalo tak, že se na vygenerovaná čísla z $[0, L-1]$ použila náhodná permutace $0..L-1$. (Motivací pro tuto verzi bylo to, abychom měli verzi, kde velikost pravděpodobnosti bude nezávislá na pozici v daném intervalu.)

Poznámka. Ještě bych chtěl podotknout, že rozlišení na tyto tři varianty přichází na řadu až při testování, nagenerovaná testovací data jsou v základní verzi.

5.2 Generování

5.2.1 Použitý generátor

Pro generování čísel s rovnoměrným rozdělením (ať už diskrétním nebo spojitým) byl použit generátor Mersenne Twister ([23]), generování čísel s geometrickým rozdělením bylo provedeno ve statistickém programu R ([15]).

5.2.2 Seed vs. předgenerování

Důležitým požadavkem při provádění experimentů je, aby je bylo možné zopakovat, a to samozřejmě na úplně stejných datech. Pokud je testování prováděno na náhodně vygenerovaných datech, máme tedy dvě možnosti:

1. mít pro experiment určené inicializační číslo generátorů (seed) a v případě opakování experimentu ho opět použít (tím se zajistí nagenerování těch samých dat)

2. náhodná data si předgenerovat

Vybrána byla možnost číslo dvě. Její nevýhodou je sice nutnost uchovávání poměrně velkých souborů na disku, ale výhodou na druhou stranu je, že během samotného testování už nemusíme nic generovat. Navíc vytváření některých souborů s daty pro testování zahrnovalo kromě vygenerovaní dat ještě další následné operace s nimi (viz např. navzájem různé indexy – 5.2.3), a to by při samotném testování dost zdržovalo. Proti zmíněným výhodám by někdo mohl namítnout, že pokud bychom nechtěli testy opakovat, tak data budeme generovat jednou a je tedy z hlediska času jedno, jestli před testováním, anebo až při něm, ale ono je to tak, že ta samá předgenerovaná data jsou využívána ne jednou, ale vícekrát v různých testech (např. pro různé operace). Ještě bych k tomu dodal jednu praktickou výhodu, která spočívá v tom, že oddělení generování a testování přináší možnost provádět generování na jiném stroji než testování. Nutností je pak přístup s předgenerováním v případě, že nemáme možnost implementovat generování do zdrojového kódu testování.

5.2.3 Předgenerované soubory

Pro účely testování bylo nutné nagenerovat několik typů dat. Při testování nestačí provést měření na jedněch datech, je potřeba ho zopakovat na několika sadách dat. Proto bylo pro každý typ dat vytvořeno více sad, ve většině případů to bylo 30, u některých, kde generování trvalo dlouho, to bylo 10.

Klíče s možností opakování

Klíče jsou celá 32-bitová čísla generovaná z rovnoměrného rozdělení. Jsou použity jako argumenty operace Insert při testování této operace. Opakování klíčů je sice možné, ale vzhledem k velikosti univerza (2^{32}) je počet případů, kdy máme vložit prvek, který tam už je, dost malý, takže výsledky jsou podobné těm, které bychom dostali v případě navzájem různých klíčů.

Klíče navzájem různé (bez opakování)

Tyto klíče jsou získány tak, že nejprve je vygenerován určitý počet klíčů s opakováním (32-bitová čísla s rovnoměrným rozdělením) a pak jsou odstraněny duplicity. (Při požadovaném počtu navzájem různých klíčů 20 milionů stačí generovat 20.1 milionu klíčů s duplicitami.) Navzájem různé klíče jsou použity jako argumenty operace Insert ve všech případech kromě testování samotné operace Insert (tedy testování úspěšného Member, neúspěšného Member, Delete, kombinace operací). Důvodem pro použití navzájem různých klíčů v těchto případech je to, že potřebujeme říct, kolik prvků chceme mít vložených před dalšími operacemi, a je jednodušší to zajistit, když jsou klíče navzájem různé. (Pokud by nebyly navzájem různé, tak by to šlo samozřejmě taky, ale bylo by to o dost komplikovanější.) Navíc jsou soubory s navzájem různými klíči natolik velké, aby po využití části souboru pro naplnění tabulky zbylo ještě dostatek klíčů, které poslouží jako argumenty případné operace neúspěšný Member (neúspěšnost je zajištěna právě vzájemnou různosti klíčů). To, že jsou pro neúspěšný Member jako argumenty užívány navzájem různé klíče, ničemu nevadí, protože záleží pouze na tom, ve kterém řetězci budeme hledat.

Náhodná čísla z (0,1)

Tato čísla jsou generována ze spojitého rovnoměrného rozdělení na $(0, 1)$. Jsou využita při strategii TS(p). Při této randomizované strategii se s pravděpodobností p použije MTF a s pravděpodobností $1 - p$ TS. To je implementováno tak, že se vezme nagenerované číslo x z $(0, 1)$ a pak:

- pokud $x < p$, je použito MTF
- pokud $x >= p$, je použito TS

Indexy

Indexy slouží jako indexy do pole s prvky vloženými do tabulky, a to při operaci úspěšný Member. Kromě indexů s rovnoměrným rozdělením byly vygenerovány také indexy s geometrickým rozdělěním, a to se čtyřmi koeficienty geometrického rozdělení – 0.001, 0.0001, 0.00001, 0.000001. Rozdělení s prvními dvěma koeficienty sloužila pro experimenty se seznamy, kde se počty prvků pohybovaly v řádech tisíců, rozdělení se zbylými dvěma při testování hašovacích tabulek, kde počty prvků byly v řádech statisíců až milionů.
V následující tabulce jsou pro ukázku některé vybrané hodnoty kvantilové funkce pro geometrické rozdělení s různými koeficienty.

Rozdělení	$F^{-1}(0.394)$	$F^{-1}(0.632)$	$F^{-1}(0.864)$	$F^{-1}(0.993)$
G(0.001)	500	1000	2000	5000
G(0.0001)	5000	10 000	20 000	50 000
G(0.00001)	50 000	100 000	200 000	500 000
G(0.000001)	500 000	1 000 000	2 000 000	5 000 000

Navzájem různé indexy

Navzájem různé indexy slouží jako indexy do pole s prvky vloženými do tabulky, a to při operaci (úspěšný) Delete. Jsou vygenerovány z výše zmíněných indexů, a to tak, že nejprve se použije operace modulo L, kde L je počet prvků, a pak se z obdrženého souboru hodnot odstraní duplicity. Pouhé odstranění duplicit by nemělo kýžený efekt, protože při testování bychom pak uplatněním operace modulo mohli zase duplicity dostat. Toto s sebou ovšem nese ten nepříjemný důsledek, že pro každý použitý počet prvků (před mazáním) je potřeba mít zvláštní soubor s navzájem různými indexy. Také je třeba poznamenat, že při odstraňování duplicit jsou odstraňovány druhé a další výskyty a jinak je pořadí zachováno, tedy do výsledného souboru se dostávají první výskyty se zachováním pořadí vůči sobě. U geometrického rozdělení to znamená, že v souborech s navzájem různými indexy budou ze začátku spíše hodnoty s vyšší pravděpodobností v daném rozdělení a ty s nižší spíš až dále v souboru.

Permutace

Permutace slouží k vytvoření náhodné verze geometrického rozdělení (více o tom v kapitole 5.1). Pro každý používaný počet prvků je samozřejmě potřeba mít zvláštní permutaci.

Poznámka. U navzájem různých klíčů a navzájem různých indexů platí, že byly vytvořeny z dat s nějakym rozdělením odstraněním duplicit. Pro jednoduchost budeme nadále používat formulaci, že i tato filtrovaná data mají toto rozdělení.

5.3 Argumenty a rozdělení u jednotlivých operací

5.3.1 Insert

Argumenty operace jsou prvky určené svými klíči, které mají rovnoměrné rozdělení. Rovnoměrné rozdělení nám pro testování stačí, protože nás zajímá pouze situace, kdy očekávaná délka všech řetězců v tabulce je stejná (to, že hašovací fce rozděluje univerzum rovnoměrně, splněno je).

5.3.2 Úspěšný Member

Jako argumenty operace se berou klíče vybrané z pole vložených prvků (klíčů) indexy s opakováním. Indexy existují jak ve verzi s rovnoměrným, tak ve verzi s nerovnoměrným rozdělením a obě možnosti byly testovány (u nerovnoměrného rozdělení je navíc ještě několik variant).

5.3.3 Neúspěšný Member

Argumenty operace jsou klíče, které mají rovnoměrné rozdělení. Nerovnoměrné rozdělení nezkoumáme, situaci z praxe dostatečně vystihuje rovnoměrné rozdělení.

5.3.4 Delete

Testování je prováděno tak, že operace Delete je vždy úspěšná (prvek určený ke smazání se v tabulce vždy nachází). Jako argumenty operace se berou prvky vybrané z pole vložených prvků navzájem různými indexy. Indexy existují ve verzi s rovnoměrným i ve verzi s nerovnoměrným rozdělením a obě možnosti byly testovány (u nerovnoměrného rozdělení je navíc ještě několik variant).

5.4 Typy testů

Pro otestování chování hašovací tabulky při použití různých strategií bylo navrženo pět testů. Ty mají různé parametry, z nichž se mnohé u jednotlivých testů opakují. Proto vysvělím význam parametrů testů raději předem, než abych se u popisu testů opakoval.

5.4.1 Popis parametrů

Velikost tabulky

Tento parametr určuje velikost hašovací tabulky. Pokud byly dělány testy se seznamy, tak byla použita velikost hašovací tabulky 1, pro testování s klasickými hašovacími tabulkami většinou 100 tisíc.

Pravděpodobnostní rozdělení

Tento parametr určuje, jestli rozdělení bude rovnoměrné, nebo geometrické.

Koeficient geometrického rozdělení

Tento parametr má samozřejmě smysl pouze, pokud je rozdělení geometrické, při použití rovnoměrného se nebere v úvahu.

Varianta geometrického rozdělení

Určuje, jestli se jedná o klasickou, inverzní, nebo náhodnou variantu geometrického rozdělení (varianty popsány v kapitole 5.1). Opět má samozřejmě smysl pouze v případě geometrického rozdělení, jinak je ignorován.

Strategie

Určuje, která strategie má být v testu použita. Na výběr jsou strategie popsané v kapitole 3.

Koeficient pro TS(p)

Tento parametr určuje hodnotu p u strategie TS(p). Při výběru jiné strategie je ignorován.

5.4.2 Test číslo 1 - Vkládání

V tomto testu se začíná s prázdnou tabulkou a pak se aplikuje určitý počet operací Insert. Argumenty těchto operací nemusejí být navzájem různé, může tedy docházet k případům, že chceme vložit prvek, který tam už je. Těchto případů ovšem díky velikosti univerza pro klíče není mnoho (při milionech operací jsou to pouze desítky tisíc), což vcelku odpovídá i reálné situaci v praxi, kdy také může k pokusům o vložení již obsažených záznamů dojít, ale není to častá situace.
Parametry:

- Počet operací Insert
- Velikost tabulky
- Strategie
- Koeficient pro TS(p)

5.4.3 Test číslo 2 - Úspěšný Member

Nejprve je vložen určitý počet prvků do prázdné tabulky a pak teprve začíná samotný test. V něm se aplikuje požadovaný počet úspěšných vyhledání.
Parametry:

- Počet prvků v tabulce
- Velikost tabulky
- Počet úspěšných vyhledání
- Pravděpodobnostní rozdělení (pro indexy)
- Koeficient geometrického rozdělení
- Varianta geometrického rozdělení
- Strategie
- Koeficient pro TS(p)

5.4.4 Test číslo 3 - Neúspěšný Member

Nejprve je vložen určitý počet prvků do prázdné tabulky a pak teprve začíná samotný test. V něm se aplikuje požadovaný počet neúspěšných vyhledání.
Parametry:

- Počet prvků v tabulce
- Velikost tabulky
- Počet neúspěšných vyhledání
- Strategie
- Koeficient pro TS(p)

5.4.5 Test číslo 4 - Delete

Nejprve je vložen určitý počet prvků do prázdné tabulky a pak teprve začíná samotný test. V něm se aplikuje požadovaný počet operací Delete (přesněji úspěšný Delete).
Parametry:

- Počet prvků v tabulce před začátkem mazání
- Velikost tabulky
- Počet operací Delete
- Pravděpodobnostní rozdělení (pro indexy)

- Koeficient geometrického rozdělení
- Varianta geometrického rozdělení
- Strategie
- Koeficient pro TS(p)

5.4.6 Test číslo 5 - Kombinace operací

Před začátkem samotného testu může být vložen do prázdné tabulky určitý počet prvků, ale v našich experimentech tato možnost nakonec využita nebyla a začínalo se vždy s prázdnou tabulkou. V testu se pak aplikuje postupně určitý počet (u každé operace může být jiný) operací úspěšný Insert, úspěšný Member, neúspěšný Member a úspěšný Delete.
Parametry:

- Počet prvků v tabulce před začátkem měřených operací
- Velikost tabulky
- Počet operací Insert
- Počet operací úspěšný Member
- Počet operací neúspěšný Member
- Počet operací Delete
- Pravděpodobnostní rozdělení – úspěšný Member
- Koeficient geometrického rozdělení – úspěšný Member
- Varianta geometrického rozdělení – úspěšný Member
- Pravděpodobnostní rozdělení – Delete
- Koeficient geometrického rozdělení – Delete
- Varianta geometrického rozdělení – Delete
- Strategie
- Koeficient pro TS(p)

5.4.7 Poznámka

Pokud se testovala nějaká operace vyžadující předtím vložení prvků (např. úspěšný Member) a při ní se měla používat určitá strategie, tak tato strategie se vždy použila už i při vkládání (to, že se musí použít už při vkládání např. strategie Sorted, je jasné, ale třeba strategie MTF by se teoreticky při vkládání používat nemusela).

5.5 Co se měřilo

5.5.1 Počet testů

Testy jsou rozdělené do dvou kategorií, testy při vyhledání a testy po vyhledání. Počet testů při vyhledání je definován jako *max(1, počet porovnání klíčů (při vyhledání))*, tedy je vidět, že se od počtu porovnání bude lišit pouze v případě, že počet porovnání je 0, tedy v případě, že řetězec je prázdný. Zjištění, že řetězec je prázdný, se počítá jako test, zatímco jako porovnání ne. Dobré je si uvědomit, že při neúspěšném vyhledání prvku v prázdném řetězci se provede jeden test a při neúspěšném vyhledání prvku v řetězci s jedním prvkem se provede také jeden test (ať už je tedy řetězec reprezentován jako seznam, nebo strom). Účelem testu po vyhledání je nějakým způsobem vyjádřit náročnost nekonstantních činností prováděných po vyhledání (příp. zjištění absence) argumentů operace. Těmito činnostmi jsou úpravy v TS(p) po neúspěšném Insert nebo úspěšném Member a úpravy při úspěšném Delete u strategie BVS. V ostatních případech se jedná o jednoduché (v některých případech dokonce žádné) úpravy s konstantním počtem elementárních operací, a tak tam nepřičítám žádné testy.

Poznámka. Počet porovnání na vyhledání jsme měřili také, ale ve výsledcích jsme prezentovali počty testů. Počty porovnání jsme ale použili později v kapitole 8.1.

Testy po vyhledání u TS(p)

V tomto případě se jedná o testy prováděné během hledání pozice, kam by se měl přesunout nalezený prvek. Protože v případě vložení prvku se žádný přesun dělat nebude, tak tyto testy mohou být nenulové pouze u neúspěšného Insert a úspěšného Member. Při hledání nové pozice se prochází řetězec od argumentu operace směrem zpět a sledují se historie požadavků na dané prvky (a jejich řešení). Testy pak budou označovat počet prvků, u kterých jsme tuto historii kontrolovali. V praxi to znamená, že pokud se má prvek posunout o pět pozic dopředu, bude potřeba šest testů, s výjimkou případů, že by to byl přesun na začátek řetězce – v tom případě by to bylo jen testů pět.

Testy po vyhledání u BVS

Zde se testy týkají samotného vymazání prvků při úspěšném Delete a určují počet prvků (kromě toho mazaného), které musíme při dané úpravě projít. Z toho, jak mazání u BVS funguje, plyne, že počet těchto testů bude nenulový, právě když mazaný prvek má oba dva syny. Právě v tomto případě je proces mazání u BVS trochu složitější a počet testů po vyhledání je vlastně roven délce cesty (měřeno počtem hran) od uzlu u s klíčem, který má být smazán, k uzlu v, který bude opravdu smazán a jehož klíč nahradí původní klíč uzlu u (viz popis strategie BVS – kapitola 3.3).

Poznámka. Ve výsledcích uvádím většinou počet testů při vyhledání nebo počet všech testů (tedy součet počtu testů při vyhledání a počtu testů po vyhledání).

5.5.2 Čas

Pro měření času se nabízí dvě základní možnosti – přerušení časovače nebo počítání cyklů procesoru.

Přerušení časovače

Pro správné plánování procesů potřebuje mít operační systém informace o časech procesů strávených na procesoru. Tyto časy počítá následujícím způsobem. Operační systém v určitých intervalech generuje přerušení časovače a vždy přičte daný interval procesu, který je právě prováděn. Takto získané časy jsou samozřejmě přibližné, protože proces může být přeplánován i při jiných přerušeních než jsou tato, a proto může být procesu započítán celý interval, i když běžel jen po jeho část, nebo naopak může běžet skoro po celý interval a přitom mu interval přičten nebude (bude přeplánován těsně před přerušením). Tyto chyby se však z velké části navzájem kompenzují, hlavně pokud bude měřený čas dostatečně dlouhý. Proto se tato metoda hodí především pro měření delších úloh, např. 1 s a delší.

Počítání cyklů procesoru

Mnoho procesorů obsahuje speciální registr, který se v každém cyklu procesoru zvýší o jedničku. Hodnotu tohoto čítače lze získat pomoci speciálních strojových instrukcí. Neexistuje platformně nezávislé rozhraní, na některých systémech lze využít přímo knihovní funkce, jinde je potřeba rozhraní doprogramovat s použitím assembleru.

Při použití této metody počítáme počet cyklů procesoru, které proběhly během dané úlohy, a z toho následně získáme i čas. Protože připočítáváme úloze všechny cykly, hodí se tato metoda pouze pro měření velmi krátkých úloh (jednotky ms), kdy je velká šance, že po celou dobu poběží opravdu jen daná úloha. Nevýhodou této metody je, že ji nelze použít na všech počítačích, protože některé procesory registr na počítání cyklů vůbec nemají.

V našich experimentech byly délky měřených časů v řádů stovek milisekund, proto se ani jedna z uvedených metod nehodí úplně ideálně. Nakonec jsme se rozhodli pro použití metody přerušení časovače, kterou jsme shledali jako vhodnější.

5.6 Hardwarová konfigurace

Procesor: Intel Pentium 1.86 GHz, sběrnice FSB 533 MHz, mezipaměť L2 cache 2 MB
Operační paměť: 768 MB DDR 333 MHz
Operační systém: Linux Fedora 7
Kernel: 2.6.21
Překladač: g++ 4.1.2

Kapitola 6

Výsledky experimentů - seznam

V této kapitole zhodnotíme výsledky experimentů prováděných na seznamech (pro zjednodušení používáme pro tuto část testů název „experimenty na seznamech“, i když u BVS se jedná o strom). V grafech jsou zobrazeny výsledky pro jednotlivé strategie křivkami různých barev a v legendě je popsáno, která barva odpovídá které strategii. Pro strategie jsou použity jednopísmenné zkratky – M (MTF), T (TR), P (TS(p)), N (klasická strategie), S (Sorted), B (BVS). Navíc u TS(p) je ještě (s výjimkou neúspěšného Member) určen parametr p, tedy například TS(0) má označení P_0.

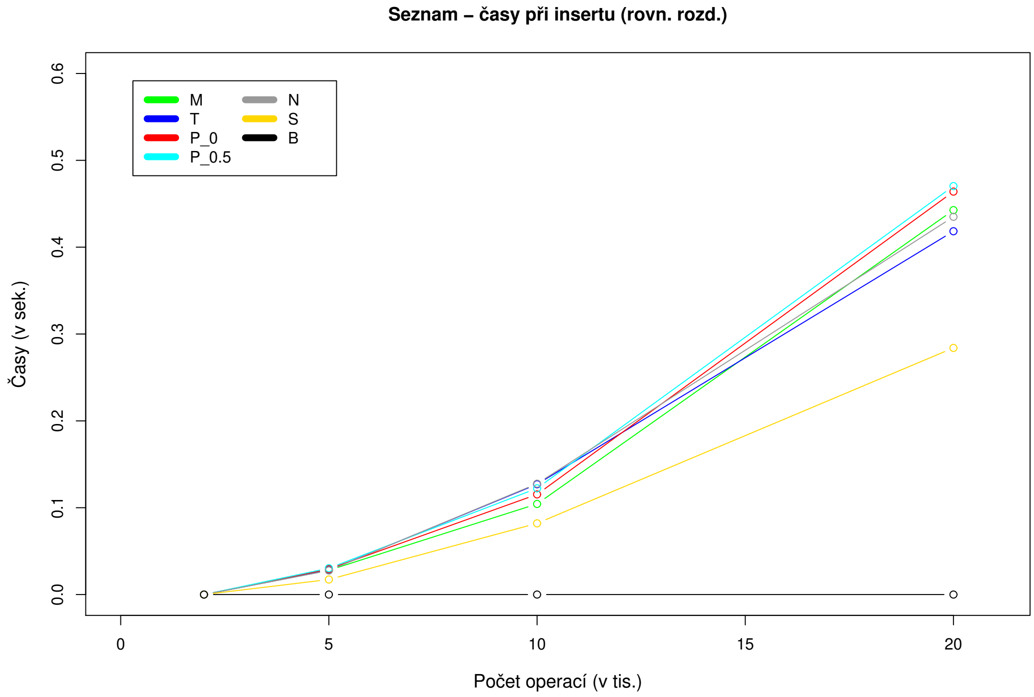
6.1 Insert

(obr. 6.1, 6.2)

Při operaci Insert začínáme s prázdným seznamem a vkládáme prvky generované podle rovnoměrného rozdělení. Může dojít i k jejich opakování, ale vzhledem k velkému rozsahu univerza a malému počtu prvků při testech se samotným seznamem (stromem) došlo při 30 měřeních na různých datech pouze k jedinému takovému případu. Proto můžeme aplikovat teoretické odhady platné pro sérii úspěšných vkládání. Ze stejného důvodu jsou také prakticky totožné grafy pro počet všech testů a počet testů při vyhledání (protože k testům po vyhledání může dojít jen u TS(p), a to nejdříve při druhém požadavku na daný prvek, kde prvním požadavkem je samotné úspěšné vložení), proto uvádíme jen první z nich.

Výsledky u operace Insert jsou nejlepší pro strategii BVS a co se týče seznamových strategií, tak pro Sorted. Všechny ostatní strategie jsou pak přibližně stejně dobré a o dost horší než Sorted. Sorted je při vkládání lepší než klasická strategie, protože se prvek nevkládá vždy na konec, ale před první větší prvek, než je on sám. Počet testů u samoupravujících strategií je stejný jako u klasické strategie, protože k neúspěšným vložením až na naprosté výjimky nedocházelo, a tak se skoro vždy procházel celý seznam.

Můžeme také porovnat některé výsledky s teoretickými odhady. Pokud se podíváme například na počty testů pro 20 tisíc operací, tak pro strategii Sorted je počet testů asi 100 milionů, u klasické strategie asi 200 milionů. To odpovídá teoretickým odhadům, protože u klasické strategie je potřeba na vložení n prvků přibližně $\frac{n^2}{2}$ operací, u strategie Sorted přibližně $\frac{n^2}{4}$ (lineární a konstantní členy



Obrázek 6.1

jsou díky dostatečně velkému n zanedbatelné). U Sorted se totiž vkládá průměrně do poloviny seznamu, zatímco u klasické strategie na konec.

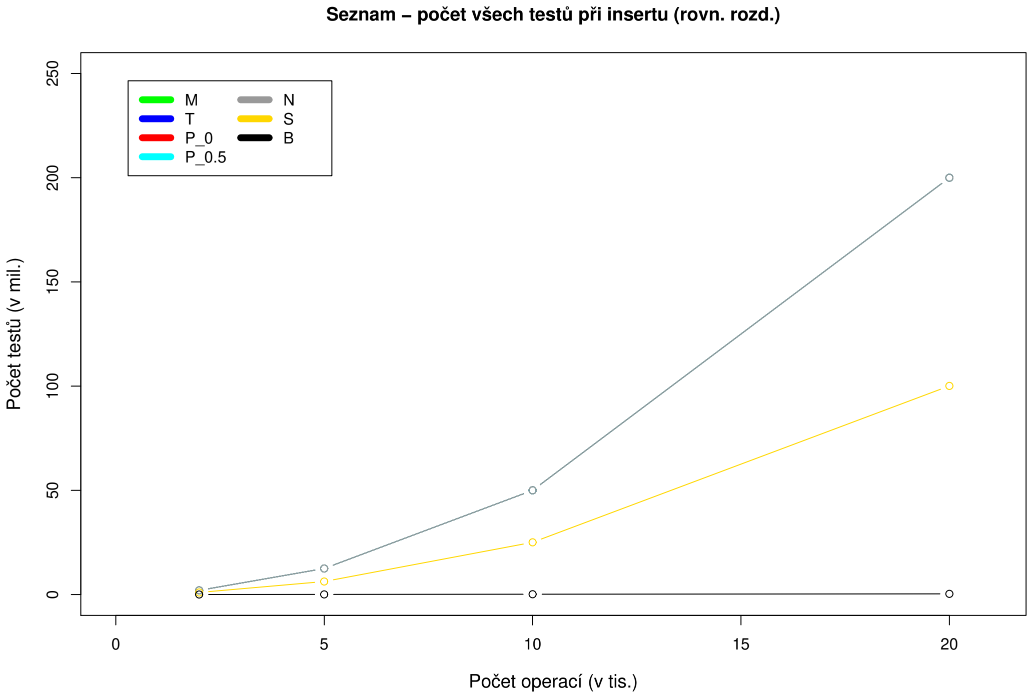
Pro BVS teoretický odhad udává, že na vložení 20 tisíc prvků je potřeba průměrně asi 339 tisíc testů a že očekávaná výška stromu je přibližně 38.2. V našem experimentu jsme potřebovali 337 tisíc testů a průměrná výška stromu (průměr byl brán z měření na různých datech) byla 32.9. Počet testů se tedy velice dobře shoduje s odhadem, u výšky je rozdíl mezi výsledkem a odhadem poměrně velký. To může být způsobeno tím, že v publikacích jsou výsledky uváděny bez méně významných členů. Například jiný odhad (viz [19]), kde není uvažován člen $\ln \ln(n)$, dává očekávanou výšku stromu v našem případě 42.7. Je tedy pravděpodobné, že i námi použitý odhad z [16] by se dal ještě upřesnit a pak by se zřejmě s obdrženými výsledky shodoval lépe.

6.2 Úspěšný Member

6.2.1 Rovnoměrné rozdělení

(obr. 6.3, 6.4, 6.5)

Při rovnoměrném rozdělení je počet testů na vyhledání stejný u všech strategií kromě BVS. Podle teoretických odhadů pro klasickou strategii by mělo platit, že očekávaný počet testů na vyhledání je roven přibližně polovině délky seznamu a výsledky tomu opravdu odpovídají, protože například pro 5000 prvků a 100



Obrázek 6.2

tisíc operací je počet testů na vyhledání asi 250 milionů. Počet všech testů se od počtu testů na vyhledání může lišit při operaci Member pouze u TS(0) a TS(0.5). Toto zde opravdu nastává, u TS(0.5) je počet všech testů vyšší než počet testů na vyhledání jen mírně, u TS(0) je tento nárůst už poměrně velký. To lze vysvětlit tím, že u TS(0.5) se s pravděpodobností 0.5 použije strategie MTF, která žádné testy po vyhledání nepřidá, zatímco u TS(0) se MTF nepoužije nikdy.

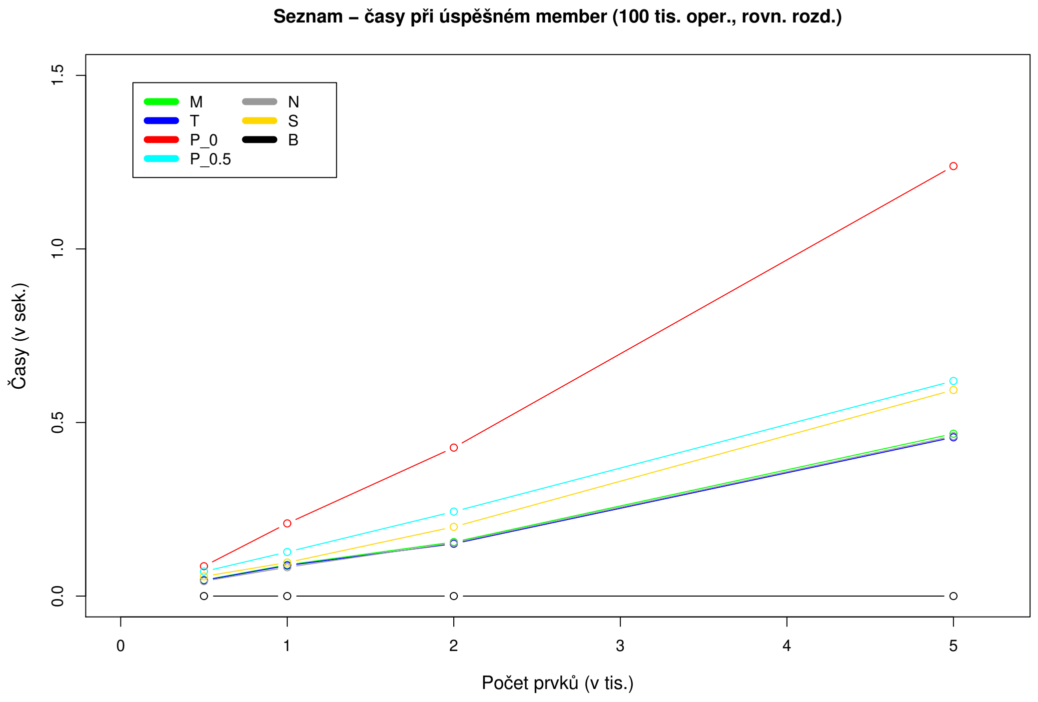
Pokud se podíváme na naměřené časy, zjistíme, že více odpovídají počtům všech testů než počtům testů na vyhledání. Jen bych poznamenal, že překrývající se křivky v grafu s časy patří strategiím MTF, TR a klasické strategii.

Porovnání BVS s ideálním stromem

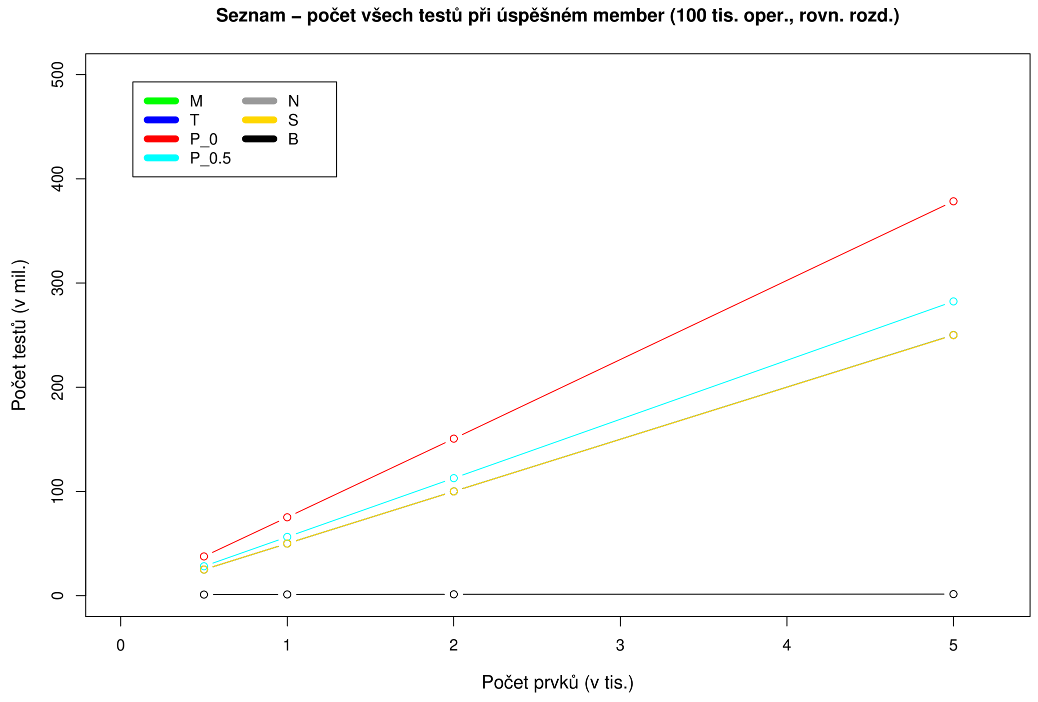
Porovnáme počet testů na vyhledání provedených při strategii BVS a počet testů, který by se provedl při stejném počtu operací, pokud bychom měli vystavěný ideálně vyvážený strom. Počet prvků bude 5000 a počet operací 100 tisíc. Ideálně vyvážený strom s 5000 prvky vypadá tak, že má zaplněných prvních 12 hladin a v té 13. je ještě 905 prvků. Za předpokladu úspěšného vyhledávání podle rovnoměrného rozdělení (tedy že pravděpodobnost vyhledávání každého prvku ve stromě je stejná) je očekávaný počet testů na vyhledání roven

$$\frac{(\sum_{i=1}^{12} 2^{i-1} \cdot i + 905 \cdot 13)}{5000} \approx 11.4.$$

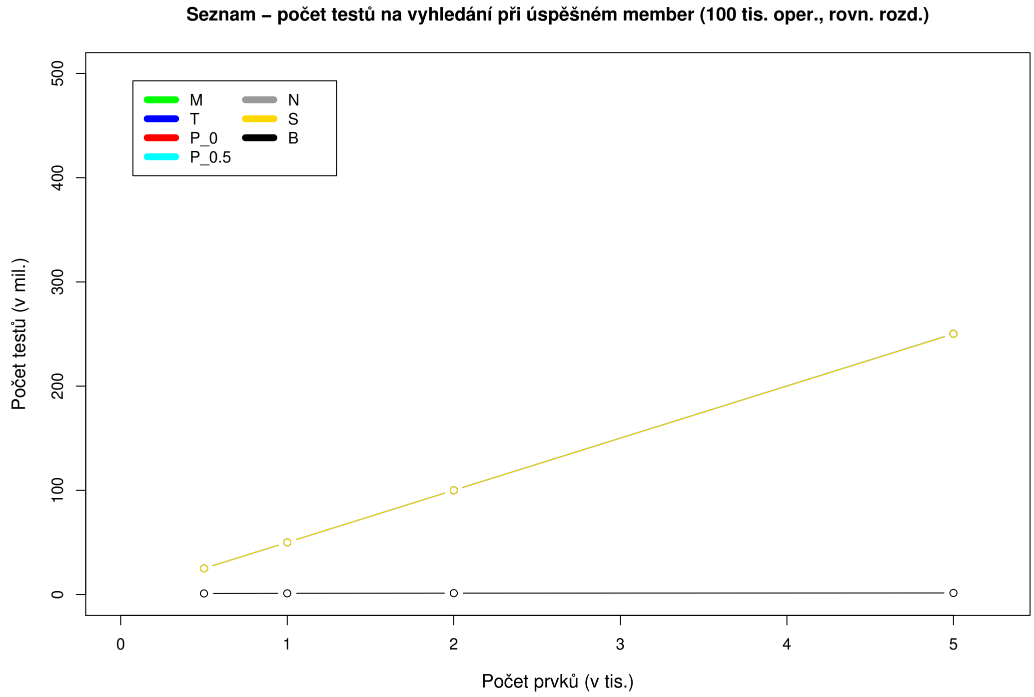
Přitom při použití BVS bylo vykonáno asi 1.508 milionů testů, což znamená průměrně 15.1 testů na jedno vyhledání. Při použití BVS se tedy provedlo asi



Obrázek 6.3



Obrázek 6.4



Obrázek 6.5

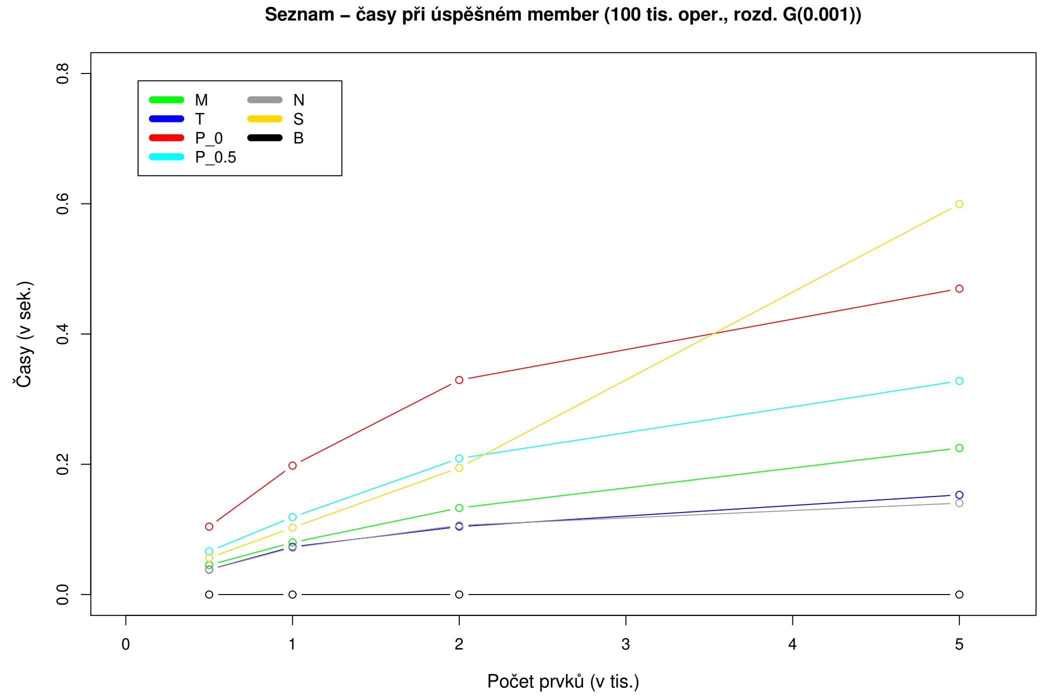
o 32.5 % testů víc, než jak by tomu bylo v ideálním stromě, což je vcelku dobrý výsledek, když vezmeme v úvahu, kolik práce ušetříme tím, že neděláme žádné vyvažování.

Ještě můžeme náš výsledek porovnat s teoretickým odhadem pro náhodné BVS. Podle toho je pro BVS s 5000 prvky očekávaný počet testů při úspěšném vyhledávání podle rovnoměrného rozdělení asi 15.19 testů. Výsledek našeho experimentu se tedy s teoretickým odhadem docela dobře shoduje.

6.2.2 Geometrické rozdělení klasické

(obr. 6.6, 6.7, 6.8)

Při tomto rozdělení se nejčastěji vyhledávají prvky vkládané ze začátku. Nejvýhodnější strategií je tedy klasická strategie, která ponechává prvky celou dobu v ideálním pořadí vzhledem k danému rozdělení. Stejně dobře se chová i TR (křivky počtu testů se překrývají), u které se pořadí prvků na začátku vyhledávání liší pouze tím, že prvně vkládaný prvek je úplně na konci. Navýšení počtu testů tímto způsobené je v grafu neznatelné a křivky se překrývají. Strategie MTF má na začátku vyhledávání nejnevýhodnější pořadí prvků, ale díky úpravám seznamu se dostanou často dotazované prvky dopředu a počet testů není o moc horší než u klasické strategie. Úplně nejhorší výsledky jsou u Sorted (alespoň u počtu testů na vyhledání to platí jednoznačně), která na rozdíl od klasické strategie nemá prvky seřazené podle toho, jak se vkládaly, ale setříděné podle



Obrázek 6.6

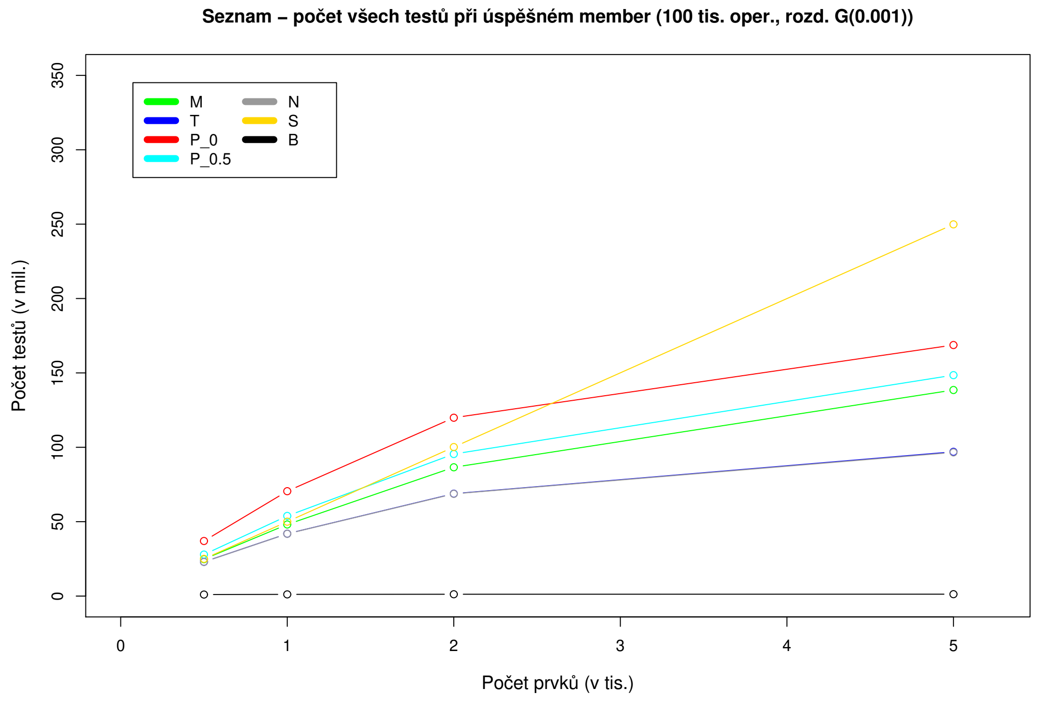
hodnoty klíče, tedy v náhodném pořadí vůči pořadí vkládání. Z toho plyne i to, že jsou pro Sorted stejné výsledky u všech typů geometrického rozdělení. Připočítání testů po vyhledání u TS(p) má za důsledek, že pro počty všech testů je strategie Sorted ta nejhorší až od určité délky seznamu a také že v počtech všech testů je pořadí trojice TS(0), TS(0.5), MTF převrácené, než jak tomu bylo u počtu testů na vyhledání.

Časy odpovídají opět vcelku dobře počtům všech testů, ale za povšimnutí stojí jeden jev, který už se dal pozorovat i u rovnoměrného rozdělení, ale zde je vidět ještě lépe. Připočítáním testů po vyhledání u TS(0) a TS(0.5) se sice více přiblížíme tomu, aby počty testů odpovídaly naměřeným časům, ale pořád to není dostatečné, neboli v časech vycházejí tyto dvě testované strategie vůči ostatním strategiím o něco hůře než v počtech všech testů. Důvodem je nejspíš to, že testování během tzv. druhého průchodu u TS(p) je časově náročnější než testování při klasickém průchodu, kdy se pouze hledá určitý prvek. Namísto pouhého porovnání klíčů se tam zjišťuje, jestli prvek splňuje podmínku z algoritmu TS(p), označenou při probírání tohoto algoritmu pro zkrácení $C(x, t)$. Tento jev bude možné sledovat i u zbylých dvou rozdělení.

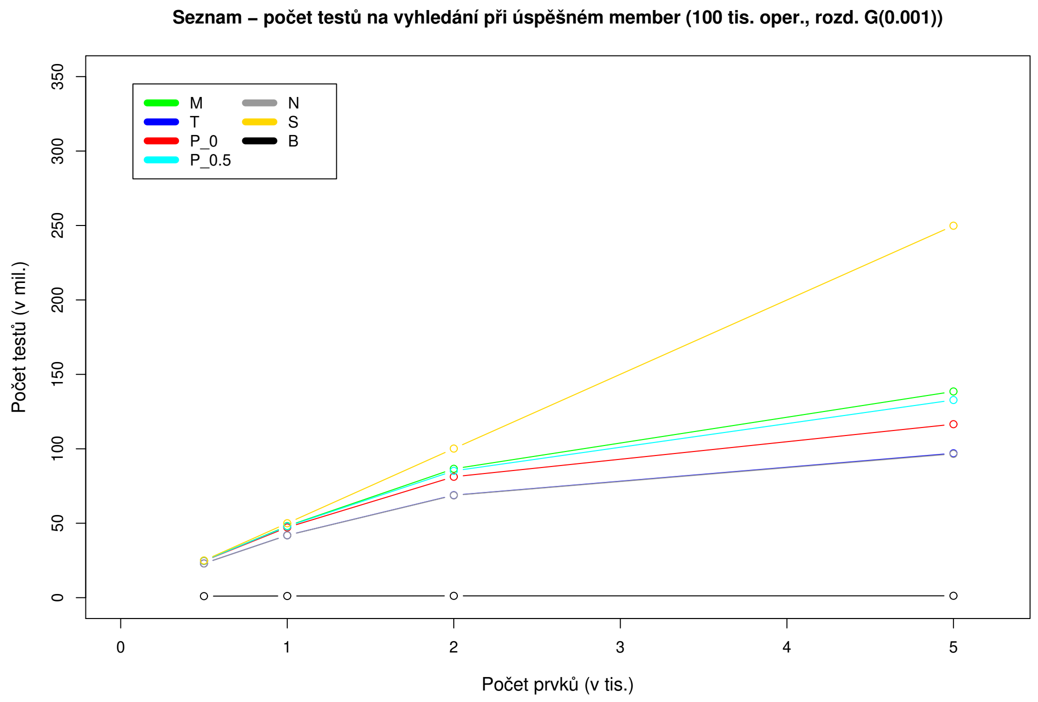
6.2.3 Geometrické rozdělení inverzní

(obr. 6.9, 6.10, 6.11)

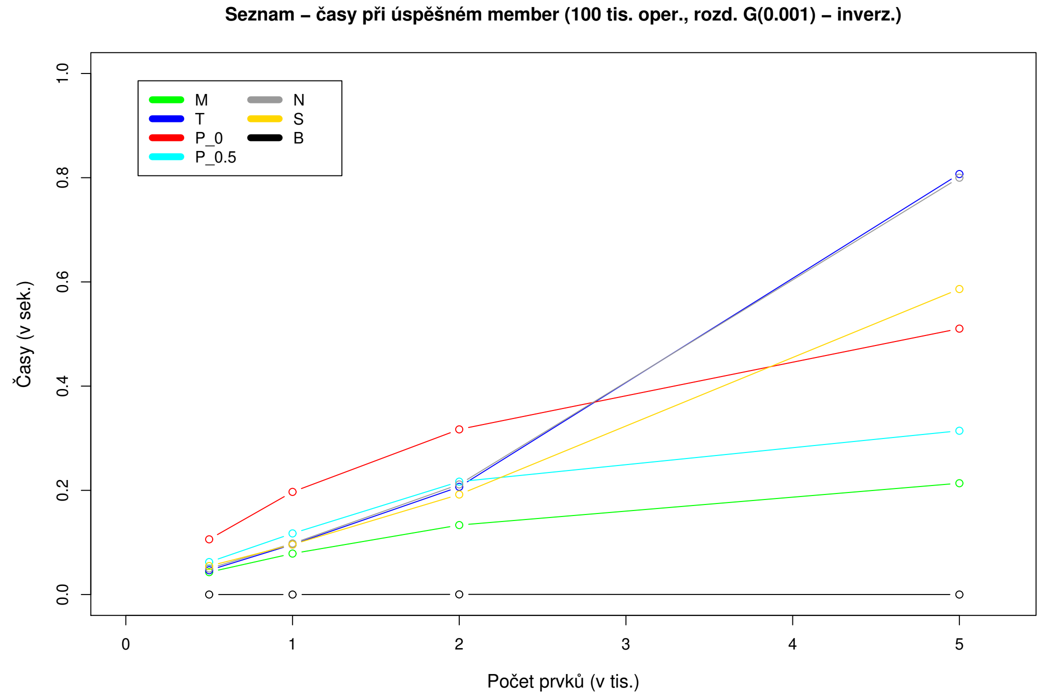
Při tomto rozdělení má klasická strategie to nejhorší možné pořadí prvků,



Obrázek 6.7



Obrázek 6.8

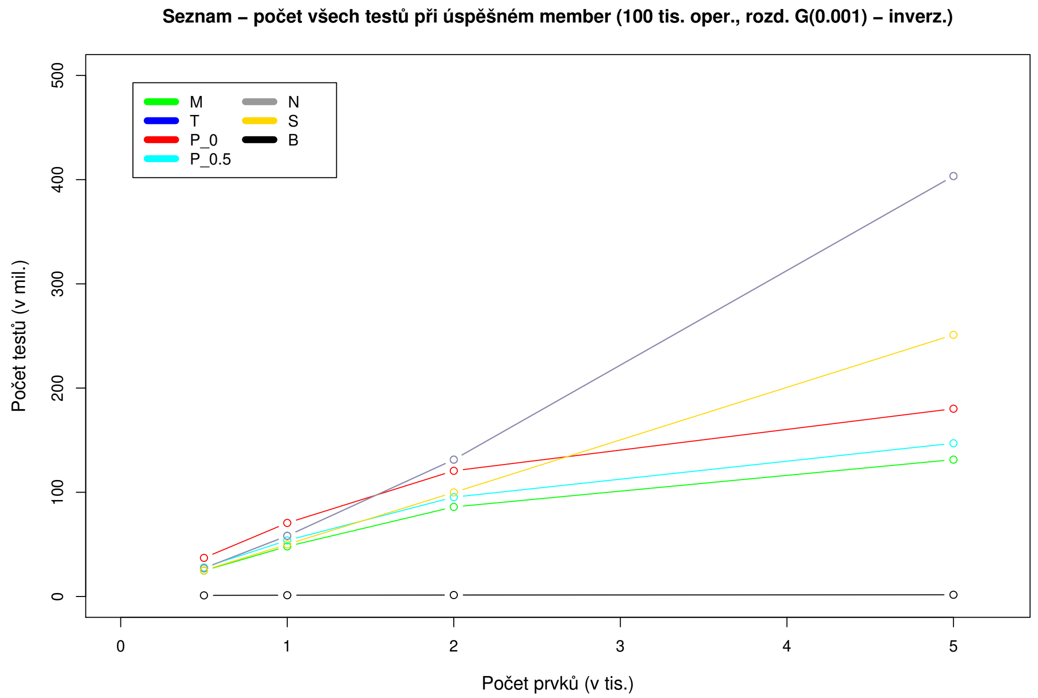


Obrázek 6.9

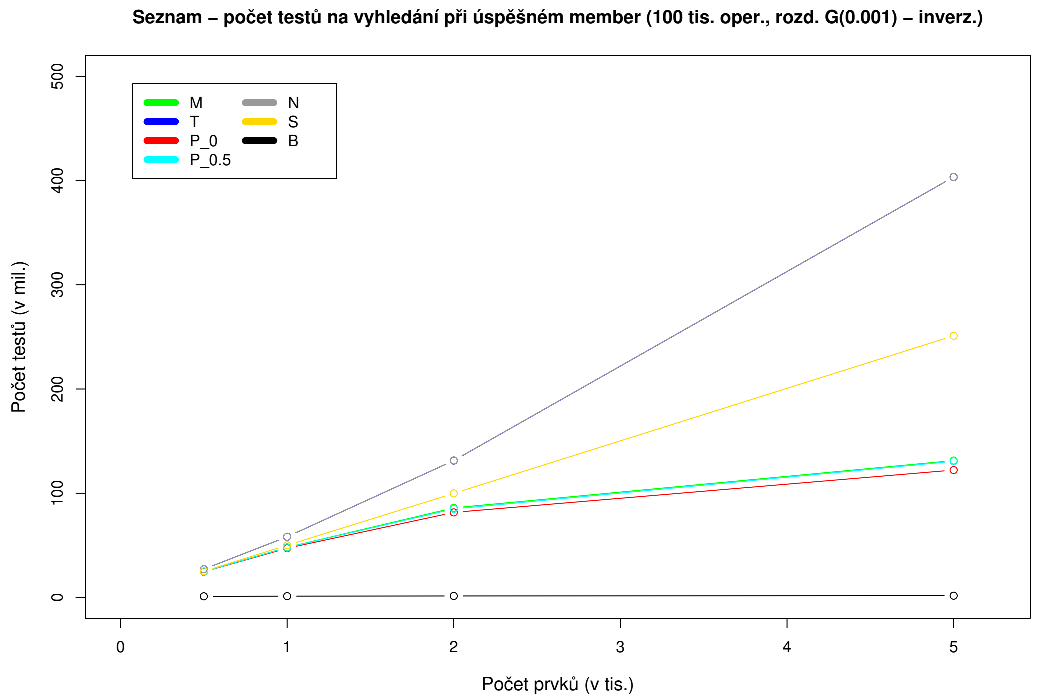
které se navíc nemění, z čehož plyne velký počet testů, které je nutné při vyhledání provést. Strategie TR má opět výsledky prakticky totožné s těmi při klasické strategii. V počtech testů na vyhledání vycházejí v podstatě shodně MTF a TS(0.5). Zajímavostí je, že strategie MTF byla poražena v testech na vyhledání strategií TS(0), přestože MTF začíná vyhledání s ideálním pořadím vzhledem k danému rozdělení (protože prvky jsou srovnané v opačném pořadí, než jak byly vkládány), zatímco TS(0) má pořadí přesně opačné, tedy naopak nejhorší. Tento jev zkusíme objasnit vzápětí.

Zajímavé je také srovnání MTF při geometrickém rozdělení inverzním a klasické strategie při geometrickém rozdělení klasickém. Pokud by totiž MTF během vyhledání už nedělala žádné úpravy, tak by dosahovala stejného počtu testů jako klasická strategie u geometrického rozdělení klasického, tedy někde kolem 97 milionů. Při použití přesunů na začátek je ale počet testů více než 130 milionů. Je to způsobeno přesuny málo požadovaných prvků na začátek seznamu, které nepřinesou žádný užitek (protože při příštím vyhledání už budou ty prvky typicky zase někde vzadu), a naopak zpomalí příští vyhledání těch často požadovaných prvků. Ze stejného důvodu je také právě MTF v testech na vyhledání horší než TS(0), protože u TS(0) se málo požadované prvky dopředu tolik neposouvají. Přitom ale na druhou stranu přesouváním těch často požadovaných se dokáže TS(0) poměrně rychle vyrovnat s nepříznivým počátečním pořadím prvků.

Zatímco v počtech testů na vyhledání byla nejlepší TS(0) a jednoznačně nejhorší klasická spolu s TR, tak u počtu všech testů a časů je nejlepší MTF a klasická s TR jsou nejhorší až od určité délky seznamu, pro kratší délky je nejhorší TS(0).



Obrázek 6.10



Obrázek 6.11

6.2.4 Geometrické rozdělení náhodné

(obr. 6.12, 6.13, 6.14)

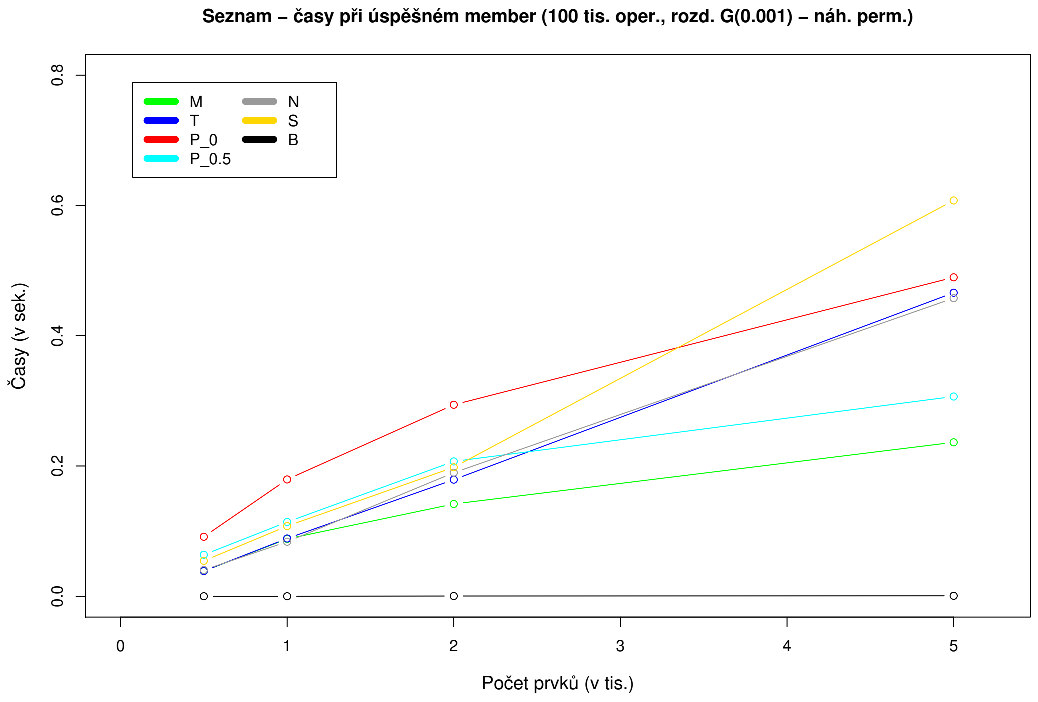
Při tomto rozdělení dostáváme podobné výsledky jako u geometrického rozdělení inverzního, pouze klasická strategie a strategie TR jsou výrazně lepší a jsou na úrovni strategie Sorted (pro tyto tři strategie se tedy křivky počtu testů překrývají). Můžeme si všimnout, že pro nesamoupravující strategie jsou výsledky stejné jako u rovnoměrného rozdělení, což je logické, protože sice jde o nerovnoměrné rozdělení, ale tím, že je nezávislé na pořadí prvků v seznamu, tak v průměru chodíme zase do půlky seznamu. U samoupravujících strategií toto samozřejmě neplatí, protože tam se mění pořadí prvků právě podle daného rozdělení, a už tedy na něm není nezávislé.

Chtěl bych se zmínit ještě o jednom jevu, který je dobře pozorovatelný v grafu pro počet testů na vyhledání. Můžeme si všimnout, že zatímco pro počet prvků 5 tisíc má skupina tří samoupravujících strategií TS(0), TS(0.5) a MTF asi o polovinu méně testů než klasická strategie, tak pro počet prvků 1 tisíc je rozdíl skoro neznatelný. To neznamená v žádném případě, že by obecně byly samoupravující strategie na nerovnoměrném rozdělení lepší třeba až od tisíce prvků, ale je to dáno konkrétním rozdělením a hlavně záleží na tom, jak velké jsou rozdíly mezi pravděpodobnostmi vyhledávání u jednotlivých prvků. V našem případě, kdy jsme použili geometrické rozdělení s koeficientem 0.001, nejsou v případě 1000 prvků rozdíly mezi pravděpodobnostmi prvků dostatečně velké na to, aby měly samoupravující strategie výrazně lepší výsledky než klasická strategie. Pokud ale bude počet prvků třeba těch 5 tisíc, tak tam už jsou rozdíly v pravděpodobnostech daleko větší (i když samozřejmě poměr pravděpodobností mezi těmi prvními tisíc prvky je stále stejný). Pokud se například podíváme na poměr pravděpodobností nejméně a nejvíce hledaného prvku, tak ten je u tisíce prvku 0.386 (pro zajímavost u Zipfova zákona je to 0.001), zatímco u 5 tisíc prvků 0.007. Proto se dá předpokládat, že samoupravující strategie by byly o dost lepší než klasická strategie i na kratších seznamech, ale potřebovaly by šikmější pravděpodobnostní rozdělení, což u geometrického rozdělení znamená vyšší koeficient rozdělení.

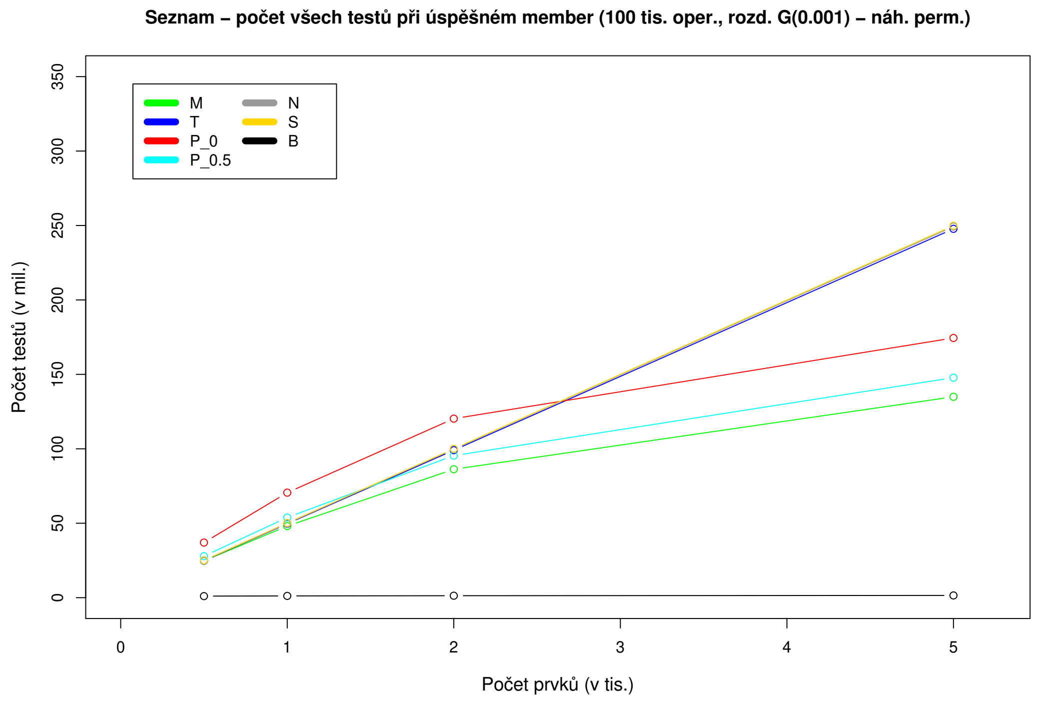
6.2.5 Porovnání s OSO

Nyní provedeme pro jednotlivé strategie porovnání počtu testů na vyhledání s počtem testů vykonaných u OSO (optimal static ordering, viz kapitola 4). Jako rozdělení použijeme geometrické rozdělení náhodné, abychom měli rozdělení nezávislé na pořadí prvků na začátku vyhledávání. (Rovnoměrné rozdělení nemá cenu zkoumat, tam je OSO stejně dobré jako vše ostatní). Budeme tedy sledovat vždy poměr počtu testů u určité strategie a počtu testů při OSO (dále jen poměr vůči OSO). Následující graf (6.15) zobrazuje tyto poměry pro jednotlivé strategie pro seznam s 5000 prvky v závislosti na počtu operací. Protože sledujeme opravdu velký rozsah počtu operací, byla použita na ose x logaritmická stupnice.

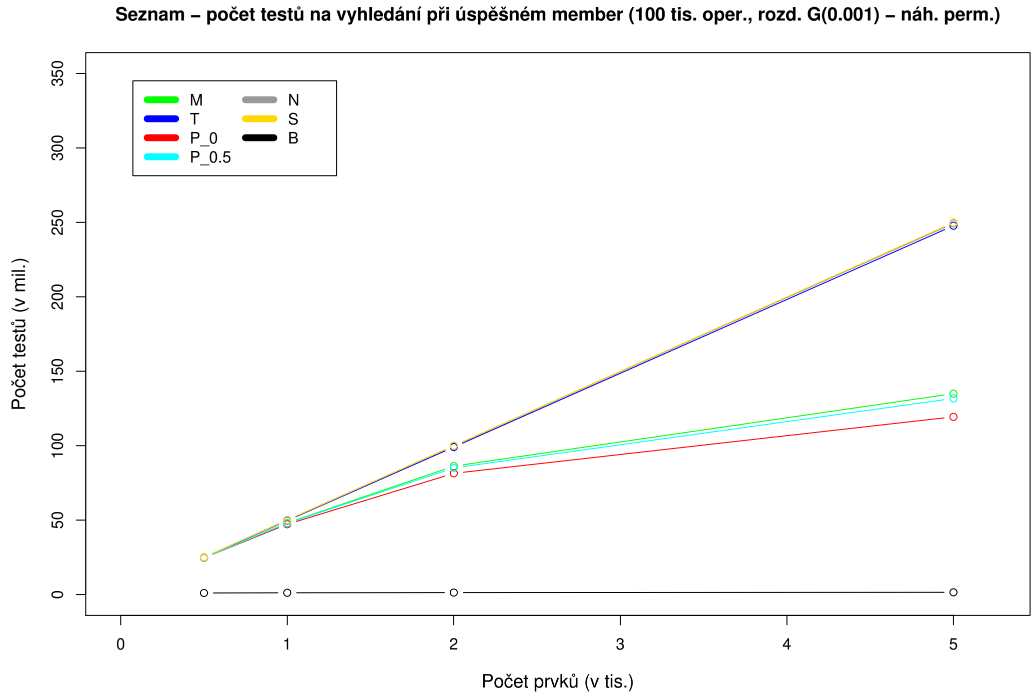
V uvedeném grafu je pro počet operací 100 tisíc vidět situace, kterou zachycuje už 6.14, tedy suverénně nejlepší BVS, pak blízko sebe skupina algoritmů TS(0), TS(0.5) a MTF a s nejhorším výsledkem skupina algoritmů TR, Sorted



Obrázek 6.12



Obrázek 6.13



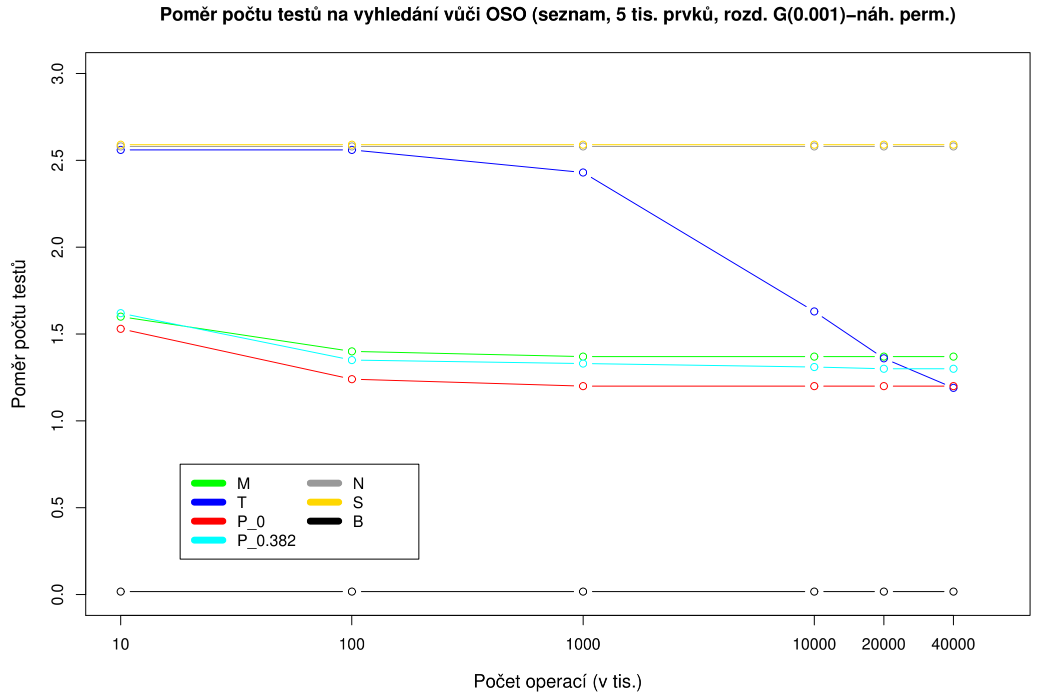
Obrázek 6.14

a klasická strategie. Při dalším zvyšování počtu operací už nedochází prakticky k žádným změnám s jedinou výjimkou, a tou je strategie TR. U té lze sledovat stále zlepšování až na úroveň nejlepší strategie TS(0). Toto chování přesně vystihuje poznámku uvedenou v kapitole 4, že TR má nízkou asymptotickou očekávanou cenu, ale konverguje k ní pomalu, zatímco MTF ji má vyšší, ale konverguje k ní velice rychle.

Nyní se podíváme na konkrétní hodnoty poměru vůči OSO u největšího testovaného počtu operací a porovnáme je s teoretickými odhady pro poměr asymptotické očekávané složitosti u dané strategie a u OSO (viz kapitola 4.1.2). U strategie MTF dává obecný horní odhad hodnotu přibližně 1.57 a horní odhad pro geometrické rozdělení je asi 1.386. V experimentech vyšel poměr přibližně 1.370, což je tedy v souladu s teorií. Pro strategii TR pak teorie říká, že by TR měla být lepší než MTF. To se také potvrdilo, protože poměr vůči OSO u ní vyšel asi 1.192. U strategie TS(0) je znám obecný horní odhad udávající hodnotu 1.34 a v testech nám vyšlo 1.196, tedy opět výsledek souhlasí s teorií. Podobně jako u MTF i zde je možná pro geometrické rozdělení odhad nižší, ale jiný odhad než ten obecný se nepodařilo najít.

6.2.6 Zhodnocení

Suverénně nejlepší strategii byla BVS, což se dalo očekávat, protože pro počty prvků v řádu tisíců už je nesrovnatelně výhodnější udržovat prvky ve stromě než v seznamu. Nadále už budeme srovnávat pouze seznamové strategie.



Obrázek 6.15

Při rovnoměrném rozdělení měly všechny strategie podobné výsledky, proto se zde nevyplatí používat nějaké alternativní strategie. Strategie Sorted měla stejné výsledky pro všechna rozdělení. U geometrického rozdělení klasického byla nejlepší klasická strategie díky ideálnímu pořadí prvků, naproti tomu u geometrického rozdělení inverzního a náhodného už byly v počtech testu na vyhledání lepší TS(0), TS(0.5) a MTF. V časech už byl lepší pro všechny délky seznamu pouze MTF, zatímco strategie TS(0.5) a TS(0) byly lepší než klasická strategie až od určitě délky seznamu. Tyto tři algoritmy byly u všech typů geometrického rozdělení přibližně stejně rychlé (rozdíly pár procent), a to asi o polovinu rychlejší než u rovnoměrného rozdělení. Na počet testů na vyhledání byla z těch tří strategií u geometrických rozdělení nejlepší TS(0), ale na počet všech testů a čas byl nejlepší MTF (vlastně si pořadí v té trojici úplně obrátily). Strategie TR vykazovala u všech rozdělení skoro stejné výsledky jako klasická strategie. Jak se ale ukázalo později, toto chování platí pro nerovnoměrné rozdělení jen do určitého počtu operací, pro velký počet operací už se výsledky strategie TR pohybují na úrovni MTF nebo jsou dokonce ještě lepší.

Pokud bychom měli vybrat nejlepší strategii pro praktické použití (a neuvažovali BVS), tak bychom asi vybrali strategii MTF, protože v počtech všech testů a časech ji překonaly akorát klasická strategie spolu s TR u geometrického rozdělení klasického, což ale není z hlediska praxe až tak důležité, protože taková situace by nastala, jen pokud bychom znali dopředu pravděpodobnosti a podle toho prvky do seznamu vkládali. Strategie TS(0) je sice lepší v počtech testů na vyhledání než MTF, ale potřebuje ještě mnoho testů na úpravu seznamu, a tak i její časy jsou o dost horší. Strategie TR má sice na nerovnoměrném rozdělení po určité době lepší chování než MTF, ale to je až po velkém počtu operací (v našem případě po desítkách milionů operací), ze začátku se dlouhou dobu chová jako klasická strategie.

Rychlá konvergence MTF a nízká asymptotická cena TR některé autory přivedla na myšlenku zkombinovat tyto dva algoritmy dohromady a využít tak výhod obou. Myšlenka je jasná – určitý čas používat MTF a pak se přepnout na použití TR. Problémem ovšem zůstává, kdy udělat ono „přepnutí se“. Bitner [9] navrhuje začínat se strategií TR po $\Theta(n)$ až $\Theta(n^2)$ vyhledáních. Sám také je autorem jednoho takového hybridního algoritmu, kombinujícího MTF a TR.

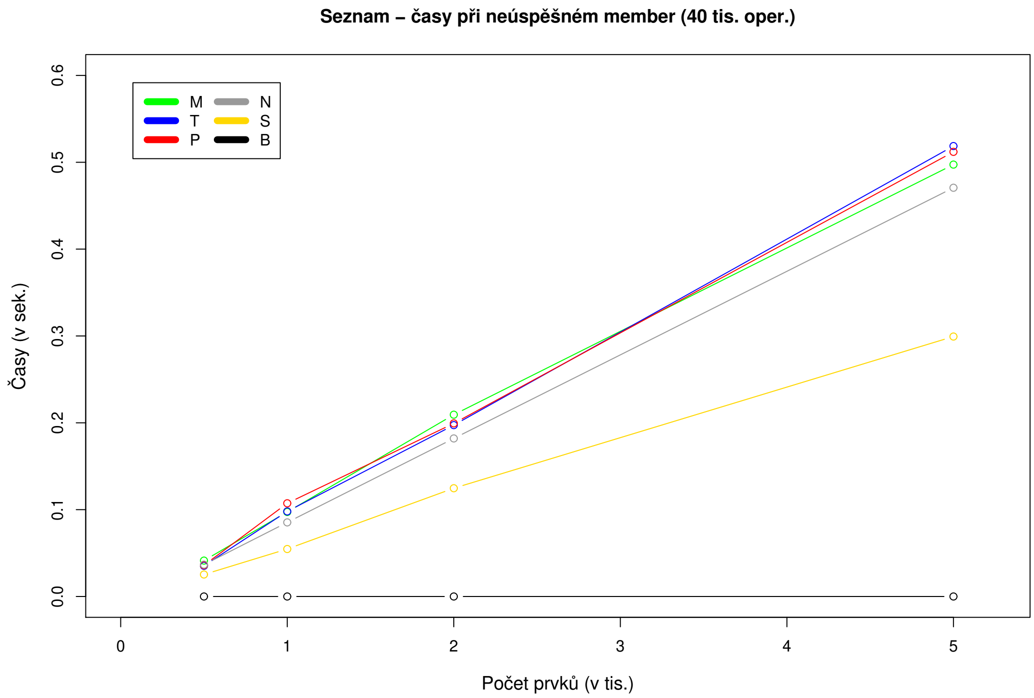
6.3 Neúspěšný Member

(obr. 6.16, 6.17)

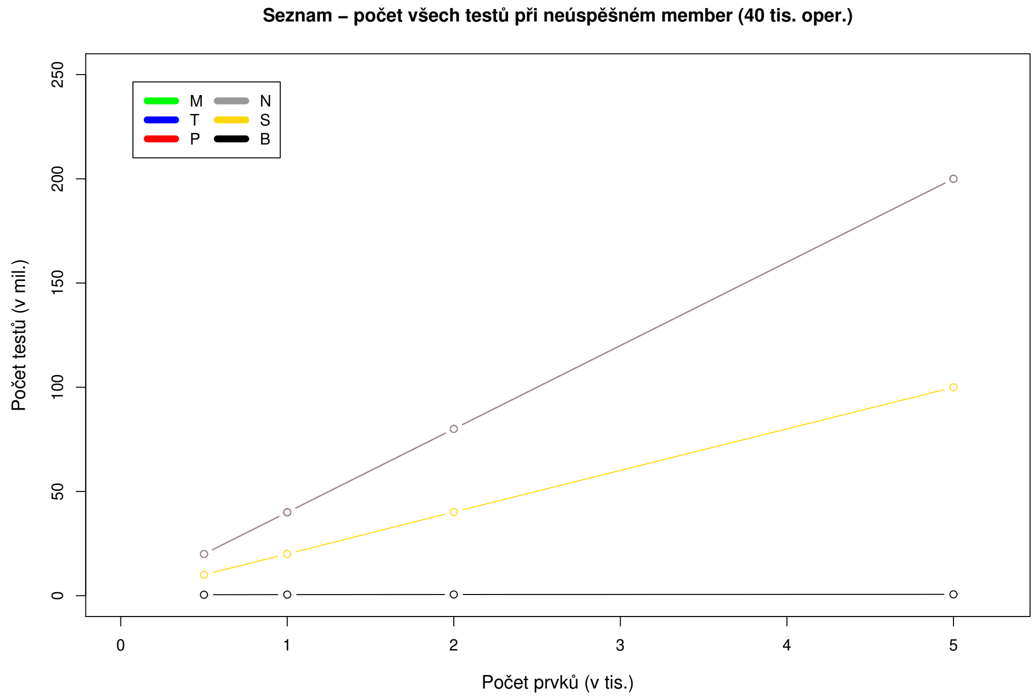
Neúspěšné vyhledání znamená u všech seznamových strategií s výjimkou Sorted projití celého seznamu. Samoupravující strategie dělají tedy přesně to co klasická strategie. Přesto jsme je pro ukázku také otestovali, pouze u TS(p) jsme otestovali jen jednu hodnotu paramteru p (a ne dvě jako u jiných operací), protože algoritmus je v tomto případě na hodnotě parametru p nezávislý (proto ani není v legendě u grafů žádný konkrétní uveden).

U neúspěšného vyhledávání jsou podobné výsledky jako u operace Insert, nejlepší samozřejmě BVS, potom Sorted a nakonec skupina ostatních strategií. Počet testů u Sorted je přibližně poloviční než u klasické strategie, protože v průměru ukončíme neúspěšné hledání v polovině seznamu.

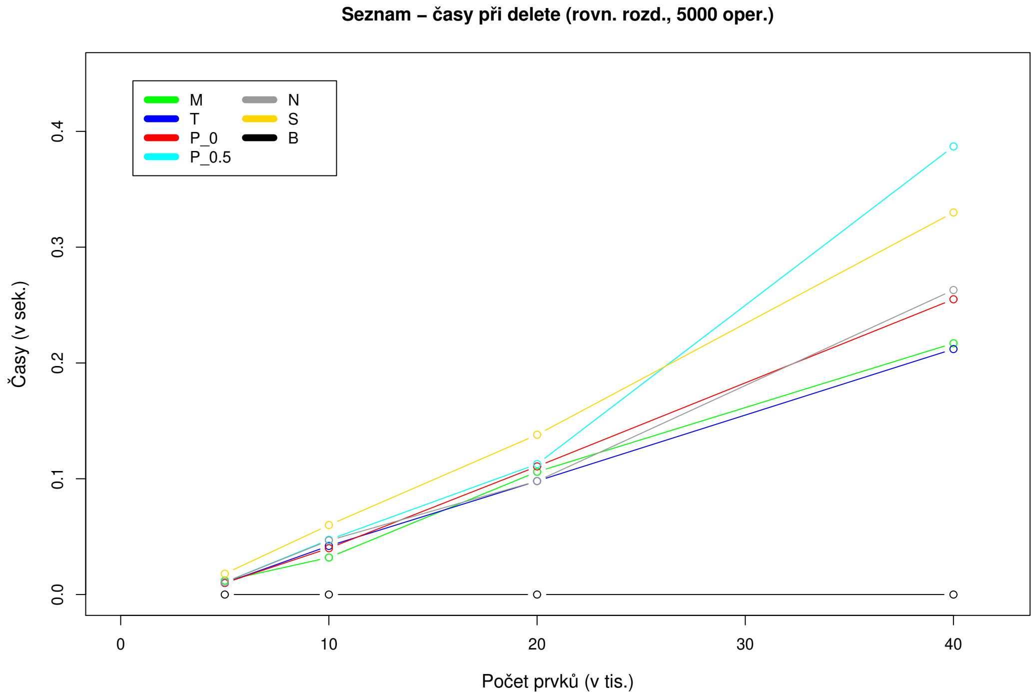
Zkusíme porovnat výsledek u strategie BVS s teoretickým odhadem. Podle toho je pro BVS s 5000 prvky očekávaný počet testů při neúspěšném vyhledávání podle rovnoměrného rozdělení asi 16.19 testů. V našem experimentu jsme při 40 tisících operací udělali asi 643 tisíc testů, což znamená průměrný počet 16.08 testů. Výsledek tedy teoretickému odhadu poměrně dobře odpovídá.



Obrázek 6.16



Obrázek 6.17



Obrázek 6.18

6.4 Delete

Poznámka

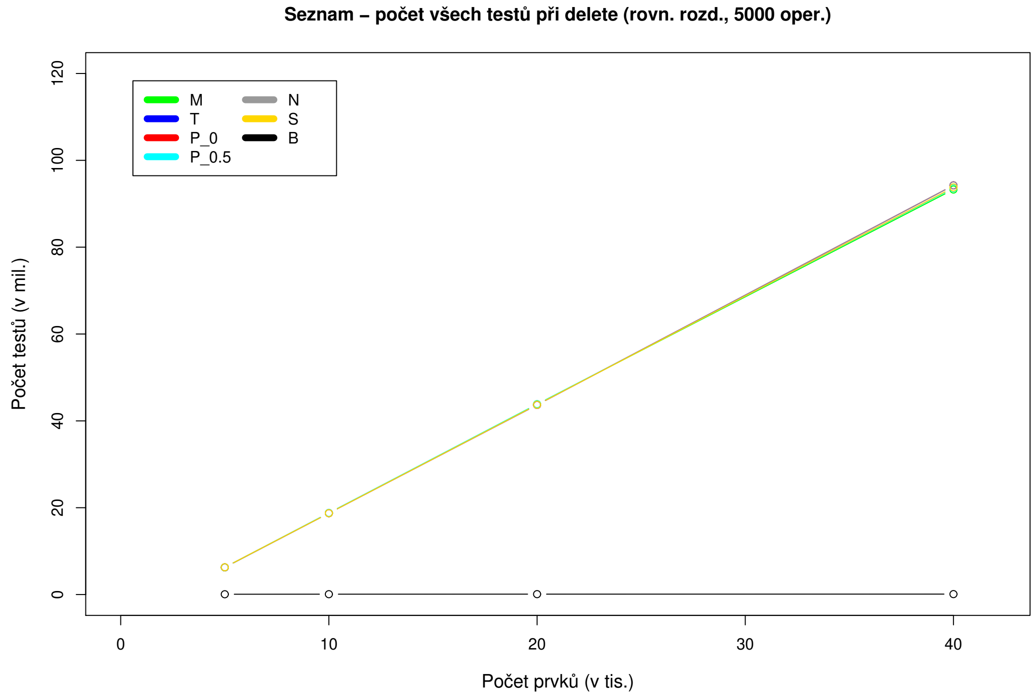
U operace Delete neuvádíme grafy s testy na vyhledání, protože jsou v podstatě totožné s grafy pro všechny testy. Testy po vyhledání se totiž u operace Delete vyskytují pouze pro strategii BVS a i zde jsou jejich počty takové, že rozdíl mezi oběma typy grafů je skoro neznatelný. Také neuvádíme výsledky pro geometrické rozdělení náhodné, protože ty jsou prakticky totožné s výsledky pro rovnoměrné rozdělení. Důvodem je to, že u operace Delete nemůže docházet k opakováním, a tak jsou z toho rozdělení jakoby vyjmuty první výskyty, a proto v případě náhodného promíchání dostaneme vlastně rovnoměrné rozdělení.

Ještě bych poznamenal, že počet prvků na ose x značí počet prvků před začátkem mazání.

6.4.1 Rovnoměrné rozdělení

(obr. 6.18, 6.19)

U rovnoměrného rozdělení vycházejí počty testů u všech strategií kromě BVS skoro stejné. Opět se tedy potvrdilo, že i když po vložení prvků je u různých strategií různé pořadí prvků, tak rovnoměrné rozdělení nakonec rozdíly vyrovná.



Obrázek 6.19

6.4.2 Geometrické rozdělení klasické

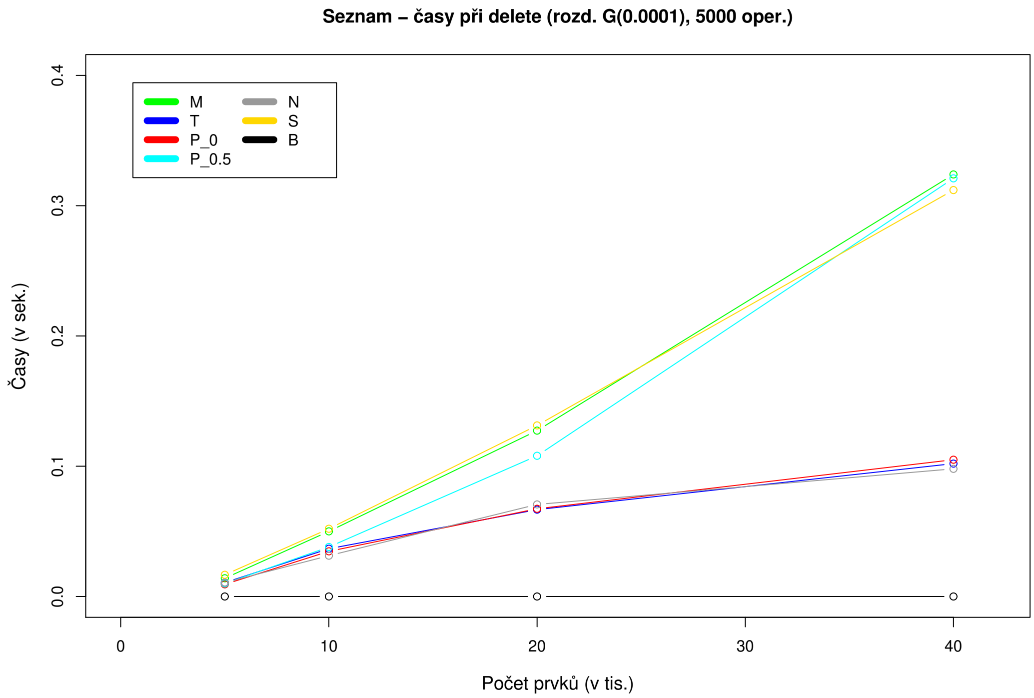
(obr. 6.20, 6.21)

Kromě BVS je nejlepší strategií klasická strategie, u níž jsou prvky na začátku mazání v ideálním pořadí vzhledem k použitému rozdělení. Stejných výsledků dosahuje i TS(0) (ta má na začátku stejné pořadí jako klasická strategie, proto je počet testů úplně stejný) a TR (ta se v pořadí prvků na začátku mazání liší pouze v jediném prvku). Naopak strategie MTF, mající na začátku mazání prvky v opačném pořadí než klasická strategie, má výsledky nejhorší. V grafu pro počet testů se kromě již zmíněné trojice strategií překrývají také křivky pro Sorted a TS(0.5).

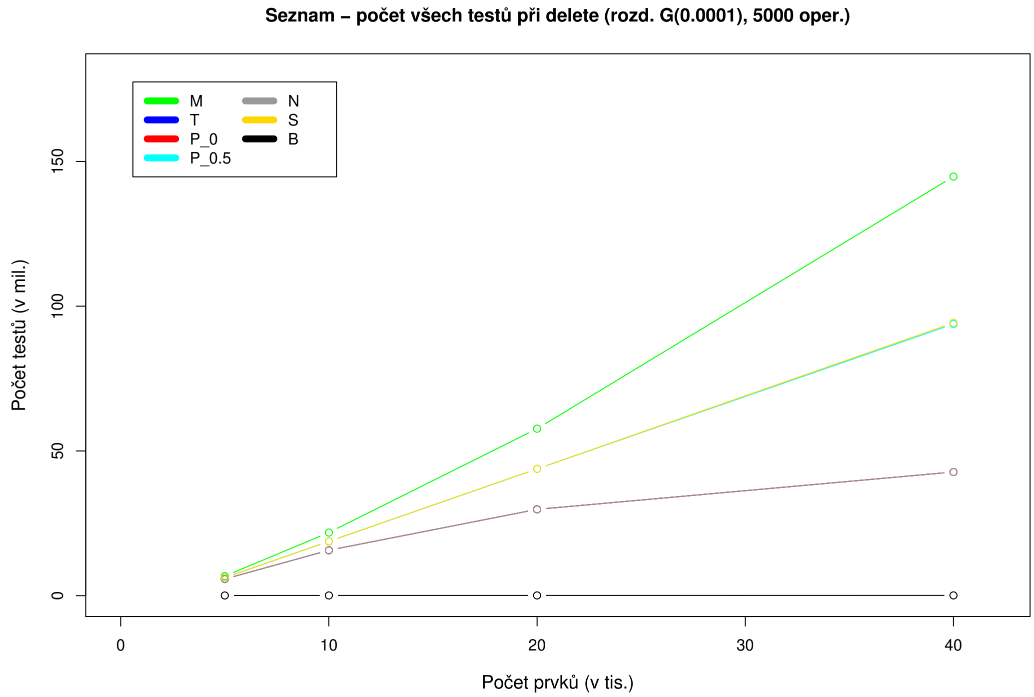
6.4.3 Geometrické rozdělení inverzní

(obr. 6.22, 6.23)

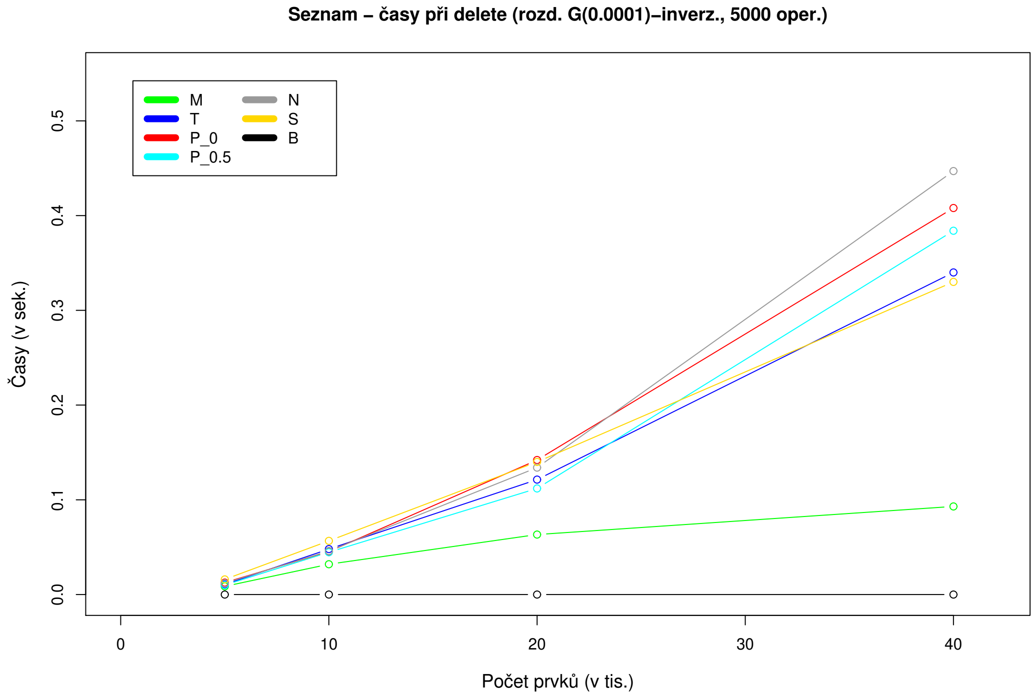
U tohoto rozdělení je nejlepší strategií MTF (nepočítáme jako vždy nejlepší BVS). Naopak strategie, které byly u geometrického rozdělení klasického nejlepší, jsou teď nejhorší. Jsou to klasická strategie, TS(0) a TR, jejichž křivky pro počet testů se překrývají. U strategie Sorted není pořadí prvků závislé na pořadí vkládání, a tak jsou výsledky pro všechny typy geometrického rozdělení stejné. Prakticky stejného počtu testů jako Sorted dosahuje stejně jako u geometrického rozdělení klasického strategie TS(0.5). Důvod je ten, že vzhledem k tomu, že se



Obrázek 6.20



Obrázek 6.21



Obrázek 6.22

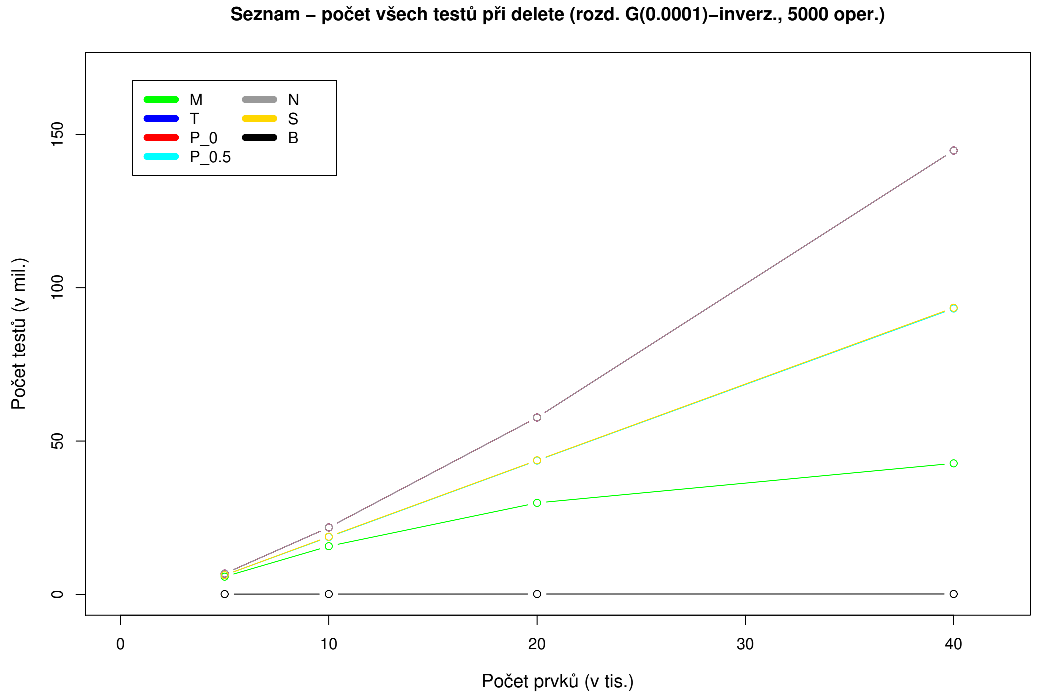
s pravděpodobností 0.5 použije MTF, při kterém se prvek přesune na začátek seznamu, a s pravděpodobností 0.5 TS, při kterém zůstane prvek na konci seznamu, tak bude pořadí na začátku mazání zřejmě přibližně náhodné vzhledem k pořadí vkládání.

6.4.4 Zhodnocení

Operace Delete nenabízí možnost využití samoupravujících strategií, protože jejich přínos bývá typicky v tom, že snižují náročnost příštího požadavku na daný prvek. Rozdíly viditelné u geometrického rozdělení klasického a inverzního byly dány různým pořadím prvků vzniklým použitím různých strategií při vkládání prvků. Ovšem toto pořadí by při praktickém použití bylo ještě často měněno vyhledáváním, proto jsou obecně důležitější u operace Delete výsledky testů pro rovnoměrné rozdělení. U toho jsou výsledky v podstatě vyrovnané, a to by byl zřejmě i závěr, co se týče operace Delete, že všechny strategie jsou pro tuto operaci přibližně stejně dobré.

6.5 Kombinace

V tomto experimentu jsme zkoušeli kombinaci operací Insert, úspěšný Member, neúspěšný Member a Delete (podrobněji popsáno v 5.4). U Insert a neúspěšného Member se používá vždy rovnoměrné rozdělení, proto v popisu grafů uvádíme jen použité rozdělení u úspěšného Member a Delete. Testovali jsme kombinaci



Obrázek 6.23

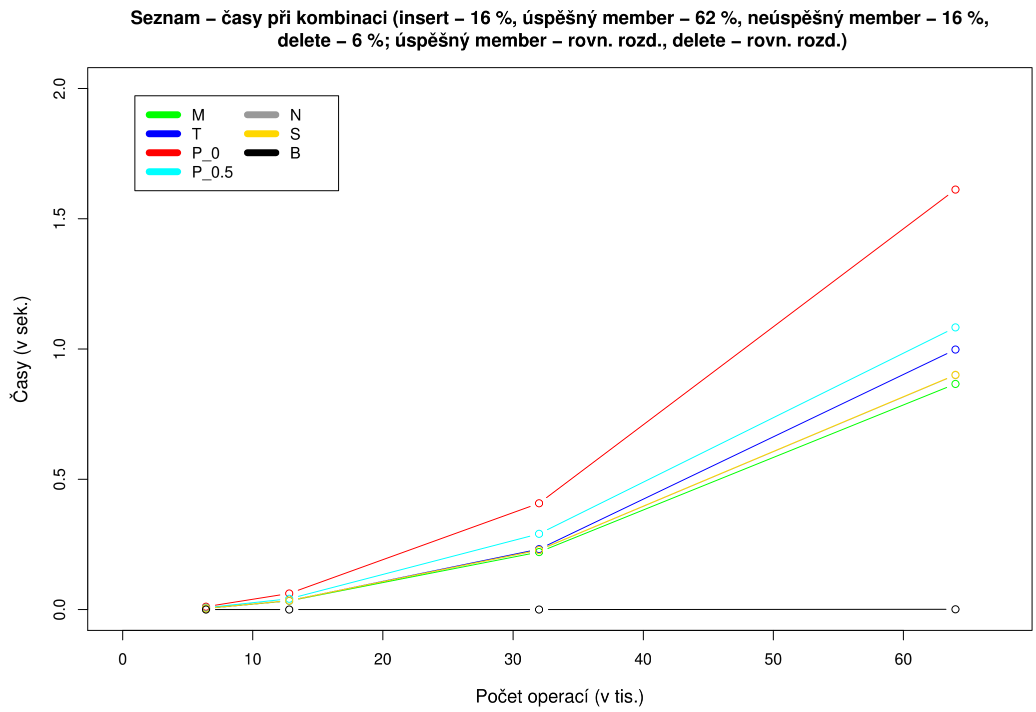
pro pět dvojic možných rozdělení u těchto dvou operací (výsledky uvádíme pouze pro čtyři, protože v jednom případě došlo ke shodě výsledků s jedním z předchozích případů, viz 6.5.3). Prováděli jsme měření pro různý celkový počet operací s tím, že bylo stále zachováno stejné procentuální zastoupení jednotlivých typů operací, a to Insert – 16 %, úspěšný Member – 62 %, neúspěšný Member – 16 %, Delete – 6 %. U operace Insert je na rozdíl od experimentu se samotnou operací Insert zamezeno opakování klíčů, tedy všechna vložení jsou úspěšná (je to proto, abychom měli při měřeních na různých datech vždy stejný počet prvků). A protože vždy začínáme s prázdným seznamem, tak počet operací Insert zároveň udává počet vložených prvků, na kterých se pak provádí vyhledávání a mazání.

Ještě bych poznamenal, že strategie TR a klasická strategie mají téměř shodné výsledky, takže ve všech grafech pro počty testů a v některých grafech pro časy se jejich křivky překrývají.

6.5.1 Úspěšný Member – rovnoměrné, Delete – rovnoměrné

(obr. 6.24, 6.25, 6.26)

V počtech testů na vyhledání je samozřejmě suverénně nejlepší BVS. Ostatní strategie mají v podstatě shodný počet testů na vyhledání s jedinou výjimkou, kterou je Sorted. Ta má méně testů kvůli tomu, že má méně testů u Insert a neúspěšného Member. Počet všech testů je od počtu testů na vyhledání rozdílný



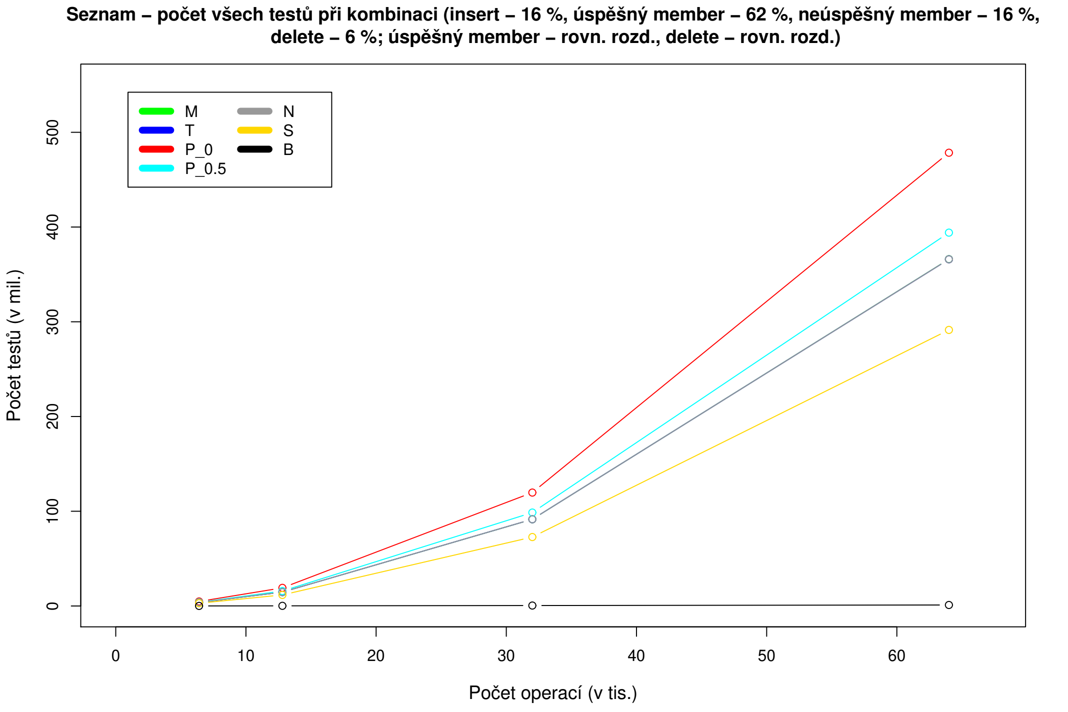
Obrázek 6.24

u TS(0), TS(0.5) a BVS (u TS(0) a TS(0.5) kvůli testům po vyhledání u úspěšného Member, u BVS kvůli testům po vyhledání u Delete). Přestože u BVS tento rozdíl není úplně zanedbatelný vzhledem k počtu testů u BVS, tak vzhledem k počtu testů u ostatních strategií zanedbatelný je a v grafu není viditelný. Podobná situace bude s testy u BVS i u dalších rozdělení, proto ji tam už komentovat nebudeme. U TS(0) a TS(0.5) je už nárůst počtu testů oproti počtům testů na vyhledání nezanedbatelný, především u TS(0) je poměrně značný. Naměřené časy pak spíš odpovídají počtům všech testů než počtům testů na vyhledání. To, že testy i časy se zvyšujícím se počtem operací rostou tak rychle, je způsobeno operací Insert (ostatní operace dávají max. lineární závislost). Ještě bych dodal, že překrývající se křivky v grafu pro časy patří klasické strategii a strategii Sorted.

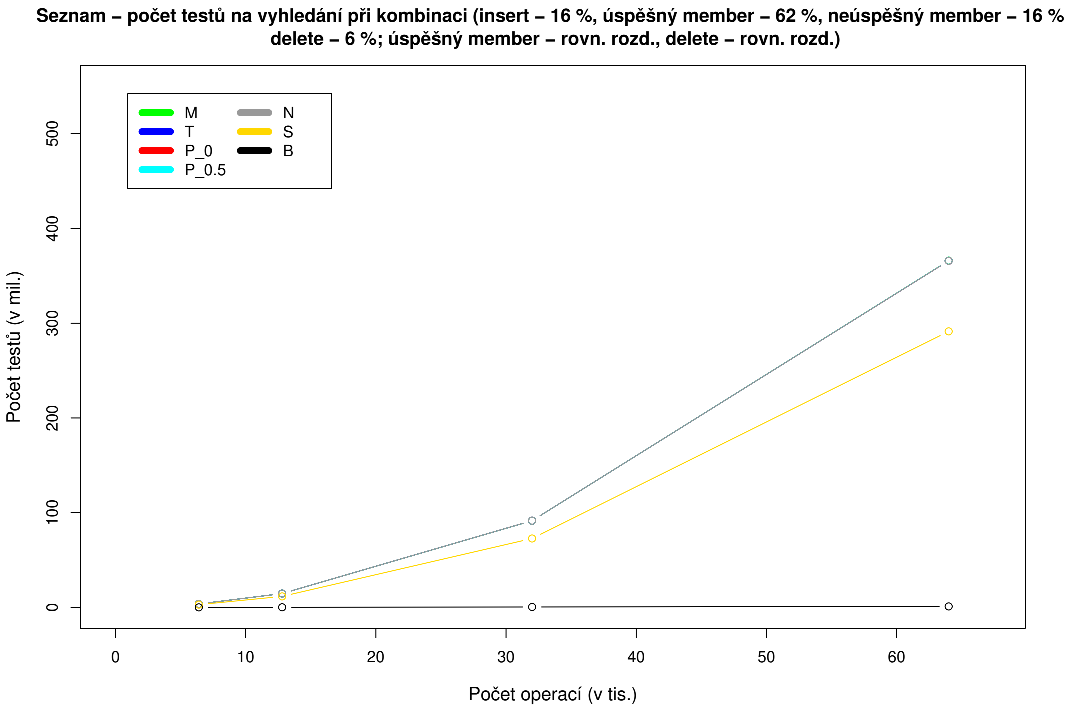
6.5.2 Úspěšný Member – geometrické klasické, Delete – rovnoměrné

(obr. 6.27, 6.28, 6.29)

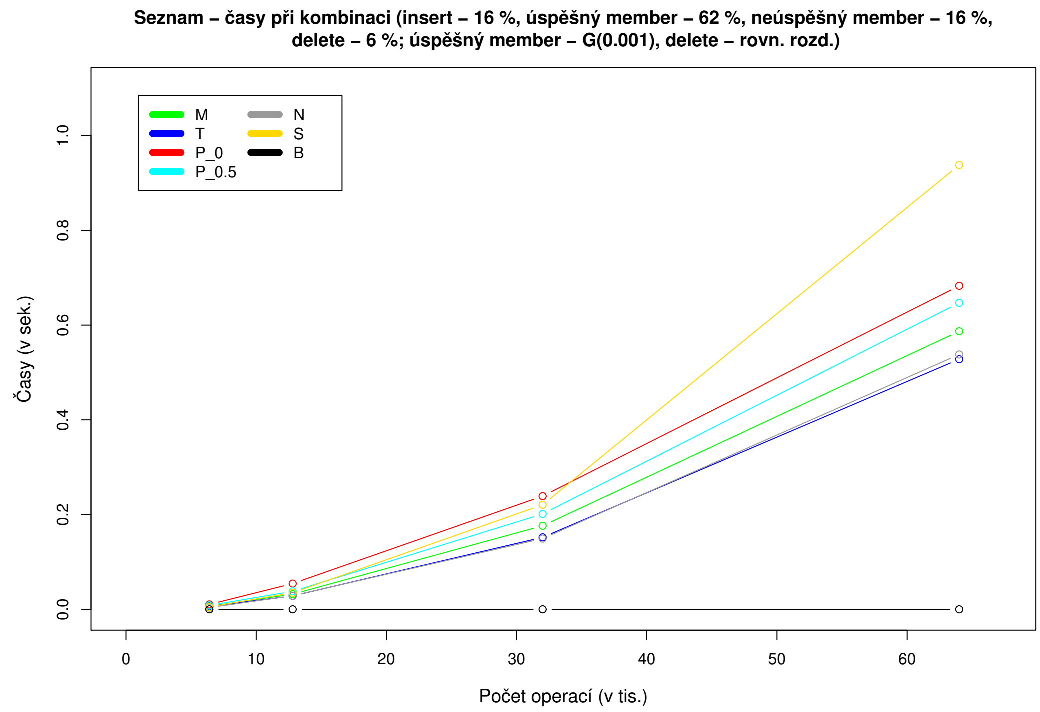
Nyní se u úspěšného Member použilo geometrické rozdělení klasické a výsledky se dost přibližují (pořadím rychlostí jednotlivých strategií) výsledkům u samotné operace úspěšný Member s tímto rozdělením. Nejlepšími strategiemi (kromě BVS) jsou z hlediska počtu testů i časů TR a klasická strategie, naopak nejhorší je strategie Sorted. Na druhou stranu odstup Sorted od ostatních strategií není tak výrazný jako u výše zmiňované samotné operace Member se stejným rozdělením. Je



Obrázek 6.25



Obrázek 6.26



Obrázek 6.27

to dáno tím, že u Insert a neúspěšného Member je Sorted naopak o něco rychlejší. Protože ale většina operací je právě úspěšný Member, tak to nevykompenzuje ten rozdíl úplně. Ještě bych doplnil, že v grafu počtu všech testů se překrývá křivka pro MTF s křivkou pro TS(0.5).

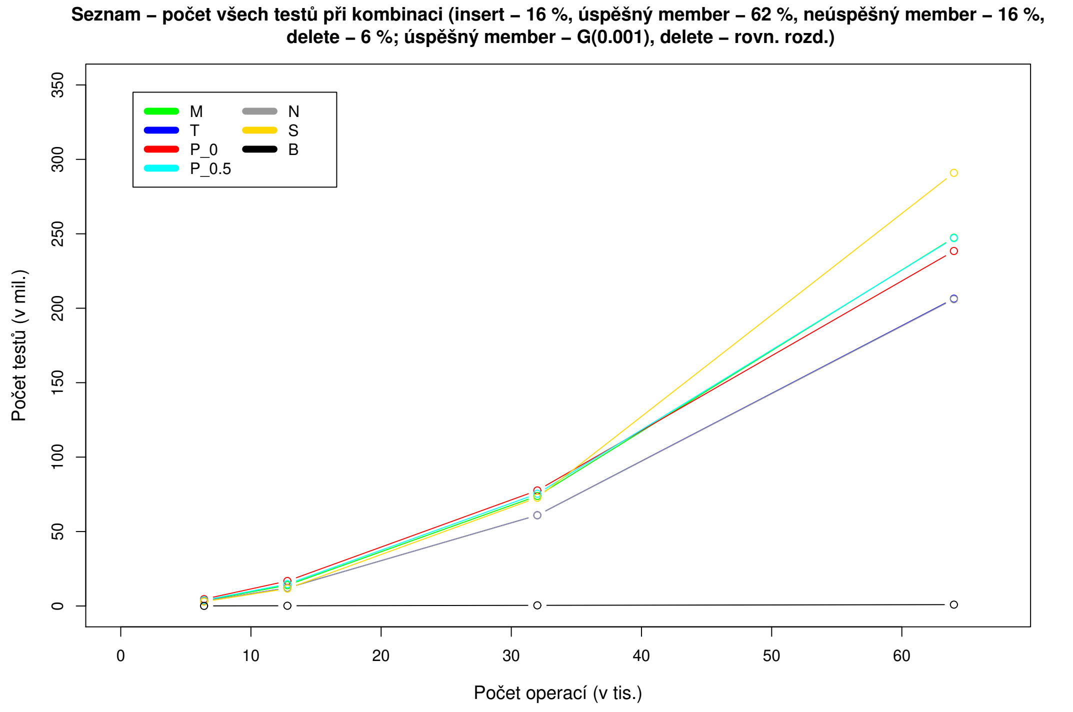
6.5.3 Úspěšný Member – geometrické inverzní, Delete – geometrické klasické

(obr. 6.30, 6.31, 6.32)

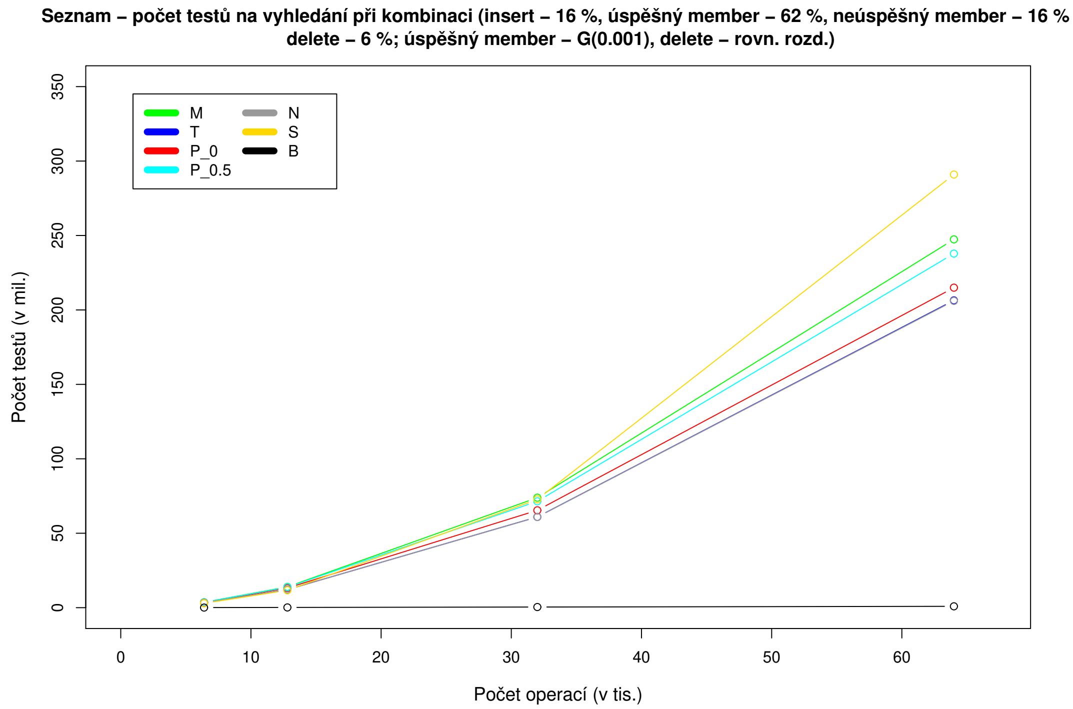
Motivací pro vyzkoušení těchto rozdělení bylo trochu simulovat možný příklad z praxe, kdy se budou vyhledávat hlavně novější prvky a mazat ty starší.

Výsledky jsou opět hodně ovlivněny typem nerovnoměrného rozdělení použitého u úspěšného Member. Používá se zde inverzní geometrické rozdělení, což má za důsledek, že nejpomalejšími strategiemi jsou TR společně s klasickou strategií, naopak nejrychlejší je (kromě BVS) strategie MTF. V počtech testů na vyhledání vychází podobně jako MTF také TS(0.5) a TS(0). Při srovnání počtu testů na vyhledání a počtu všech testů je vidět největší rozdíl u TS(0), která v porovnání se Sorted vychází na počet testů na vyhledání přibližně stejně a pro maximální testovaný počet operací lépe, zatímco v počtu všech testů vychází hůře a pro maximální testovaný počet operací stejně. Naměřené časy vcelku věrně kopírují počty všech testů.

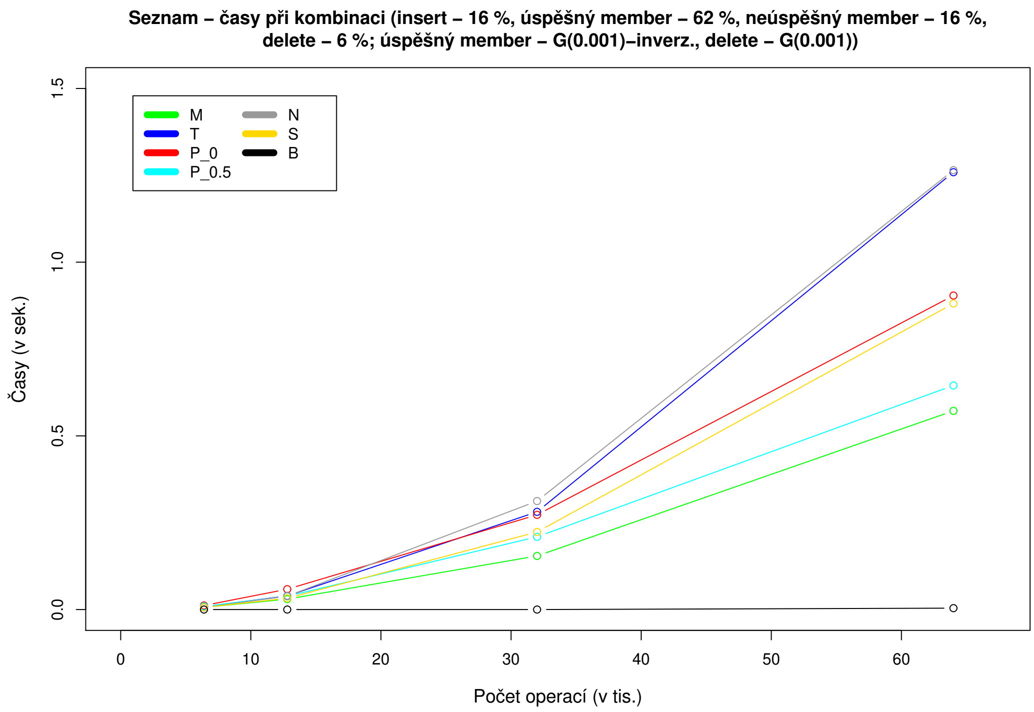
Testovali jsme také obdobnou situaci s tím rozdílem, že pro Delete bylo použito



Obrázek 6.28



Obrázek 6.29



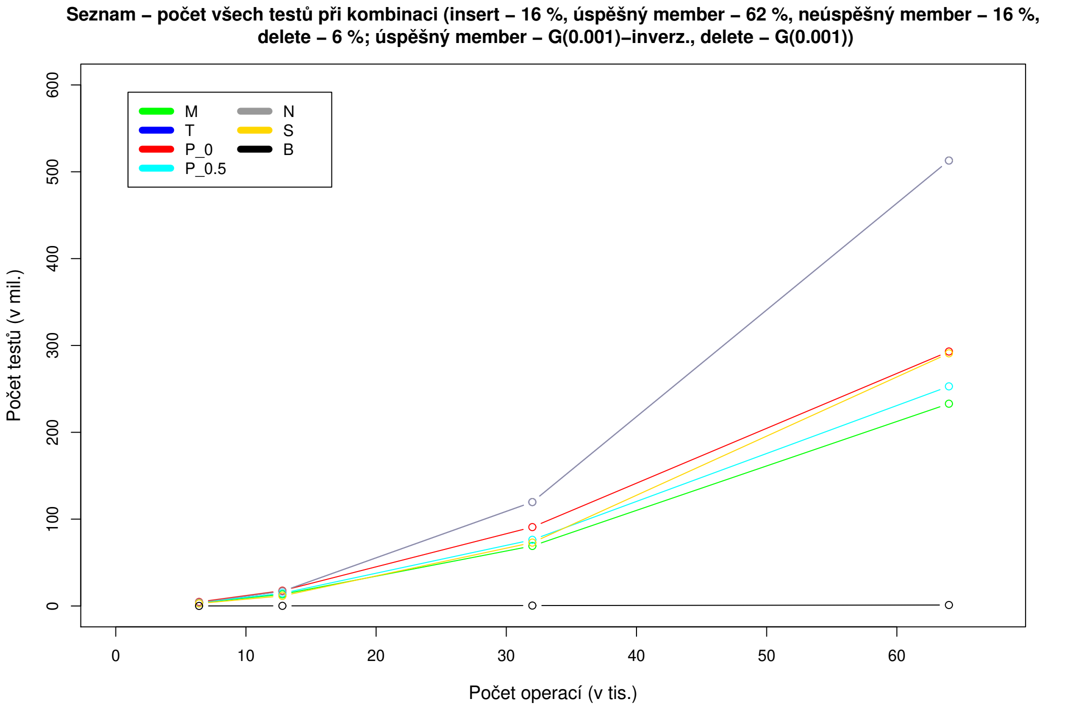
Obrázek 6.30

rovnoměrné rozdělení. Výsledky byly v podstatě totožné, proto pro ně nebudeme ani uvádět grafy. Ukázalo se, že výběr rozdělení pro Delete nemá na výsledek žádný podstatný vliv, což je ale dáno zejména malým zastoupením operace Delete v kombinaci (pouze 6 % operací).

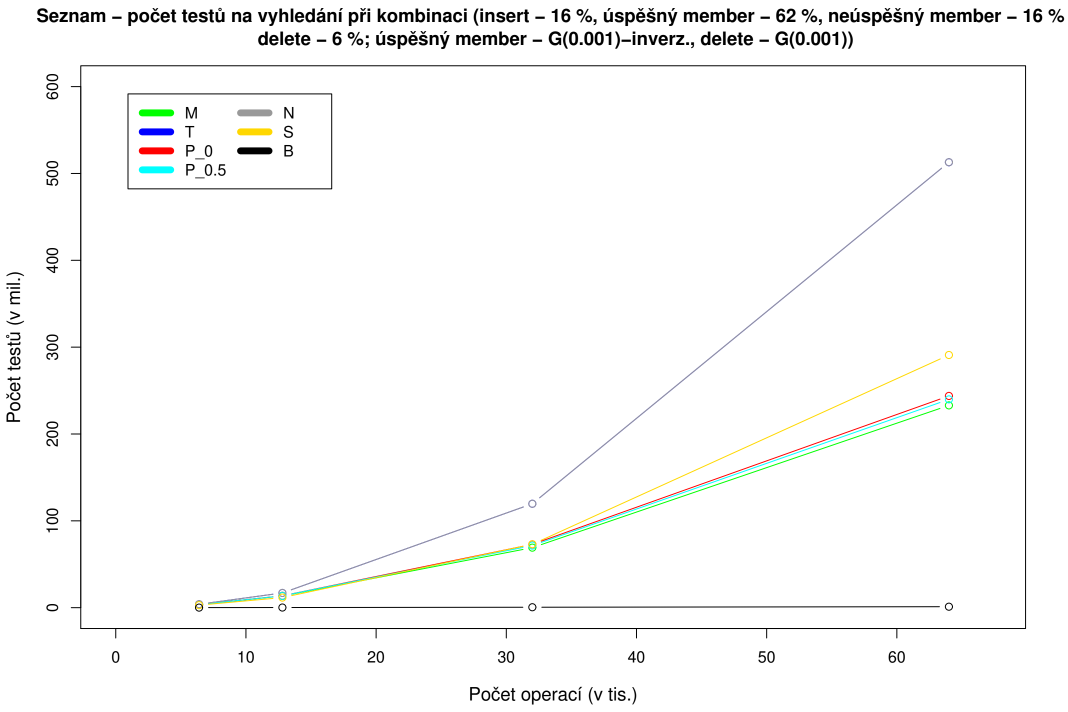
6.5.4 Úspěšný Member – geometrické náhodné, Delete – rovnoměrné

(obr. 6.33, 6.34, 6.35)

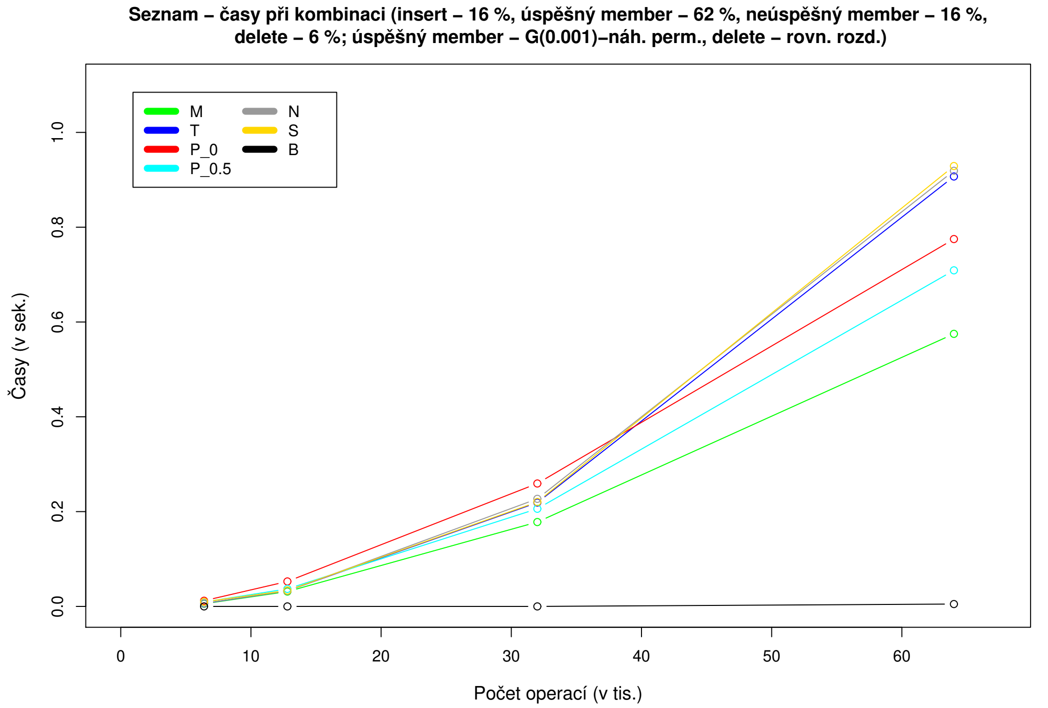
V počtech testů na vyhledání je nejrychlejší seznamovou strategií TS(0), těsně následovaná MTF a TS(0.5), které mají počty testů na vyhledání prakticky totožné. Nejpomalejší jsou naopak TR a klasická strategie (i když už nejsou o tolik horší jako v případě, kdy u úspěšného Member bylo použito geometrické rozdělení inverzní). Díky menšímu počtu testů u Insert a neúspěšného Member je Sorted lepší než klasická strategie a v počtech všech testů je pro menší testované počty operací dokonce lepší než TS(0). V počtech všech testů i časech je nejlepší seznamovou strategií MTF. V naměřených časech lze vidět drobné rozdíly oproti grafu všech testů v tom, že pro menší počet operací je TS(0) horší než klasická strategie a že Sorted je v časech na úrovni klasické strategie (křivky se v grafu překrývají).



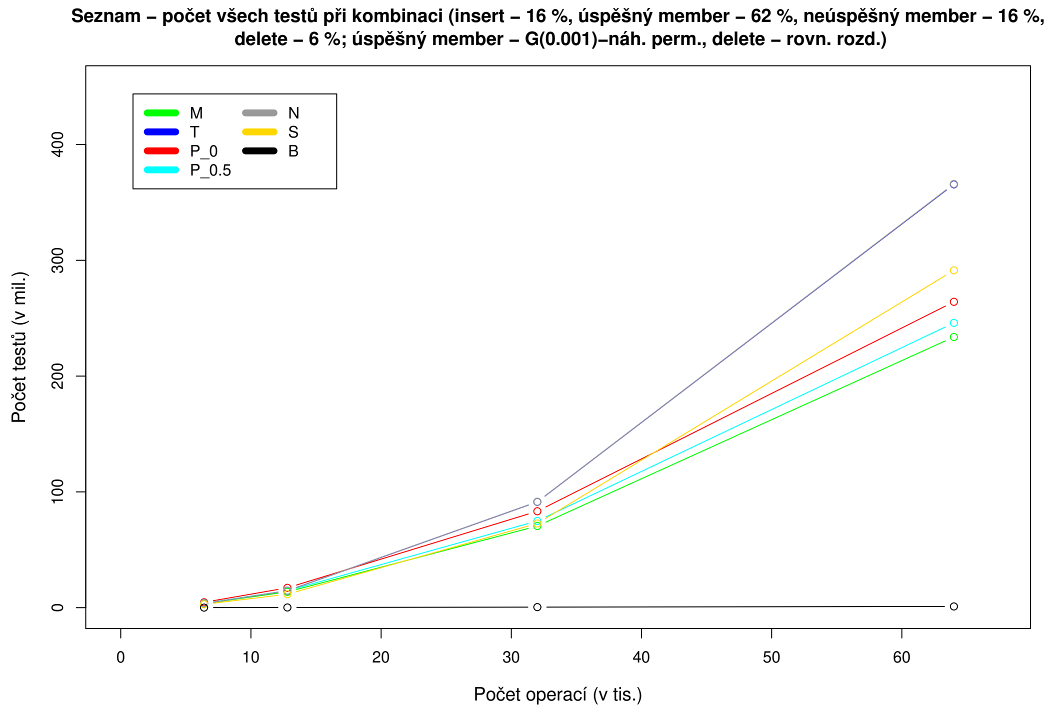
Obrázek 6.31



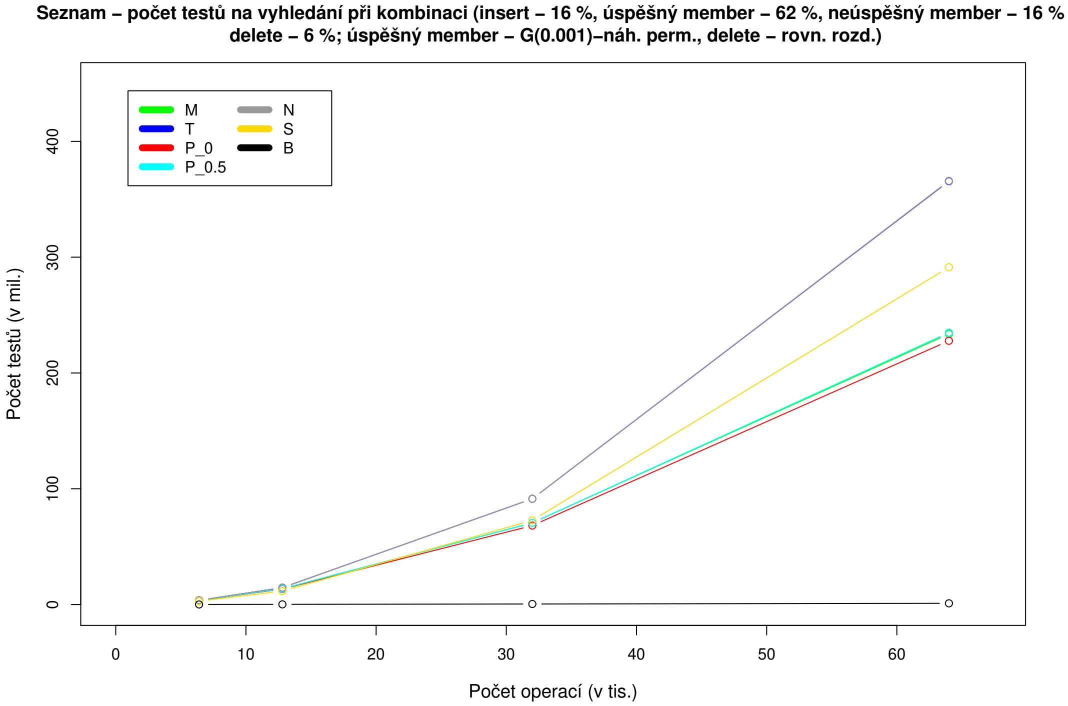
Obrázek 6.32



Obrázek 6.33



Obrázek 6.34



Obrázek 6.35

6.5.5 Zhodnocení

Výsledky u jednotlivých možností byly vždy velmi ovlivněny použitým rozdělením u operace úspěšný Member, která měla procentuelně největší zastoupení, a dost se podobají výsledkům (alespoň co se týče pořadí strategií) u úspěšného Member s příslušným rozdělením. Strategie Sorted má ale vůči ostatním seznamovým strategiím lepší chování než u samotné operace Member, protože u Insert a neúspěšného Member je rychlejší než ostatní seznamové strategie. Pokud bychom chtěli vybrat nejvhodnější seznamovou strategii, tak by to byla asi z hlediska počtu testů na vyhledání strategie TS(0) a z hlediska počtu všech testů a času strategie MTF. Samozřejmě že ale celkově nejlepší je z testovaných strategií suverénně BVS.

Kapitola 7

Výsledky experimentů - tabulka

Zatímco při experimentech na seznamech jsme pracovali s řetězci s řádově tisíci prvky, tak v tabulkách budeme mít řetězce o dost kratší. Proto bude zajímavé srovnat výsledky testů na tabulkách s analogickými testy na seznamech. Experimenty jsme dělali většinou pro různé faktory naplnění. V grafech není uváděn na ose x explicitně faktor naplnění, ale je tam uveden počet prvků, příp. u Insert nebo kombinace počet operací (které v sobě skrývají údaj o počtu vložených prvků), a faktor naplnění snadno získáme jako podíl počtu prvků a velikosti tabulky. Pro popis jednotlivých strategií v legendách grafů platí stejný systém jako u grafů pro experimenty na seznamech.

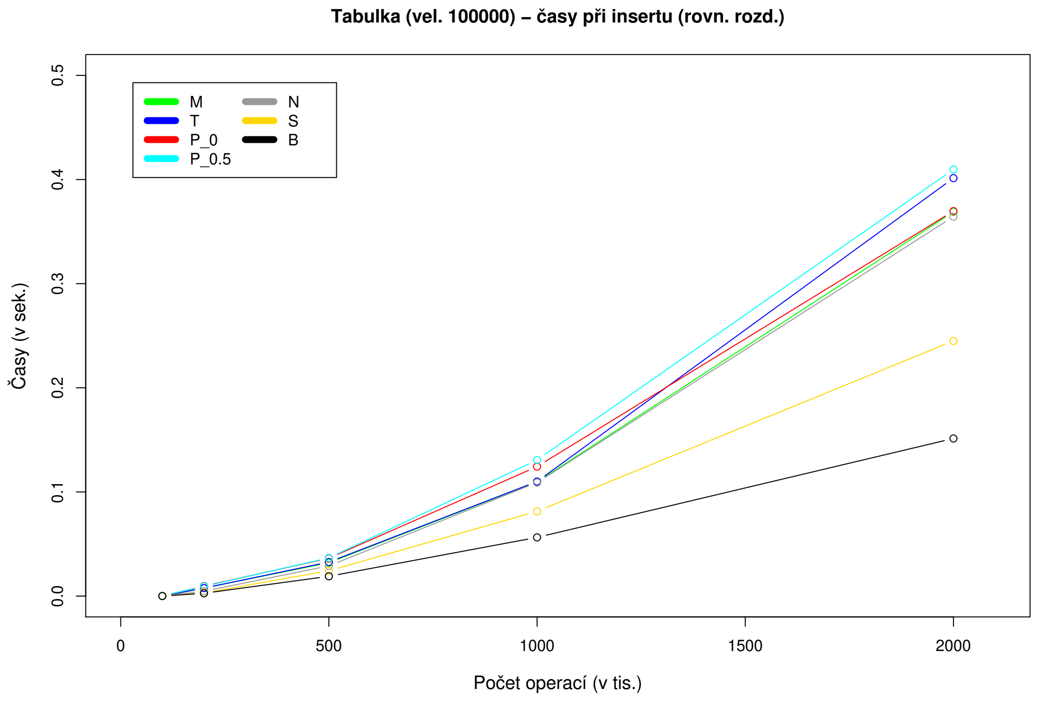
7.1 Insert

(obr. 7.1, 7.2)

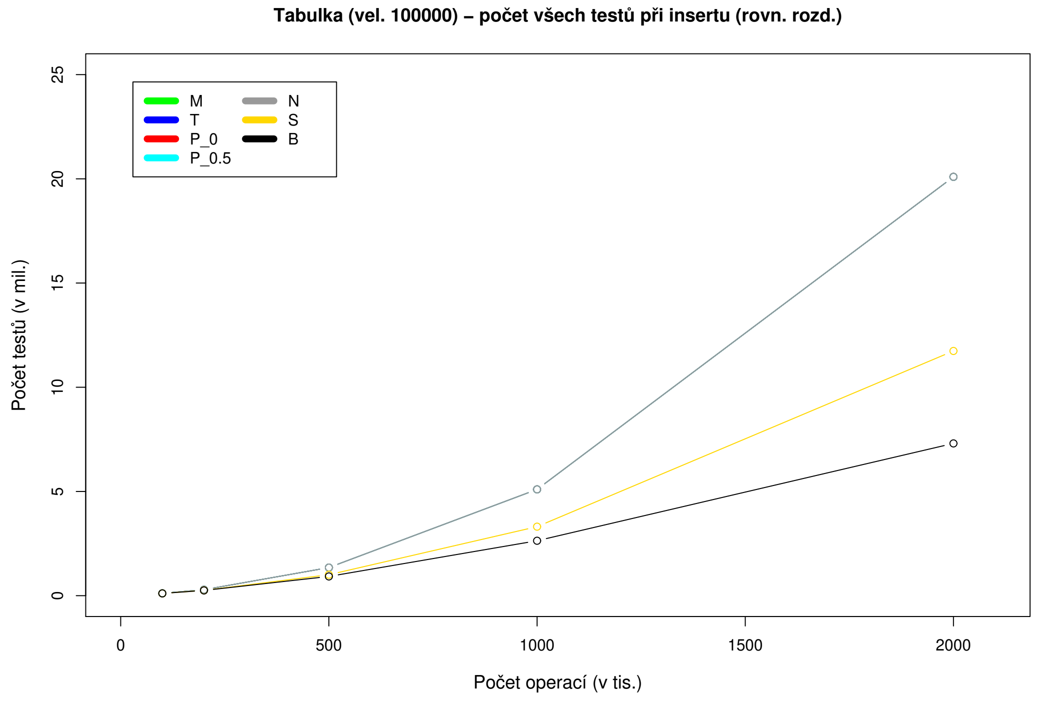
Při testování operace Insert začínáme s prázdnou tabulkou a vkládáme prvky generované na základě rovnoměrného rozdělení. U operace Insert se testy po vyhledání mohou vyskytnout pouze u TS(p), a to až při neúspěšném vložení, tedy teprve až když vkládám prvek, který tam už je, a k tomu dochází opravdu minimálně (řádově tak tisíce případů), proto je graf pro testy na vyhledání prakticky totožný s grafem pro všechny testy, a tak uvádím pouze ten pro všechny testy.

Nejmenšího počtu testů dosahuje BVS, na druhém místě je Sorted a pak společně všechny ostatní strategie. To, že u seznamových strategií je pro všechny strategie kromě Sorted stejný počet testů, je dáno tím, že málokdy dojde k neúspěšnému vkládání, a tedy v naprosté většině případů se vždy projde celý seznam. Navíc ani větší počet neúspěšných vkládání by nezpůsobil nevyrovnanost počtu testů, jak bude vidět u úspěšného Member (úspěšný Member je de facto totožný s neúspěšným Insert). Menší počet testů u strategie Sorted než u klasické strategie je dán tím, že se nemusí procházet celý seznam a prvek se vloží před první větší prvek.

Počtům testů odpovídají celkem i časy, ve skupině se stejným počtem testů jsou rozdíly mezi časy jen minimální. Výsledky jsou podobné výsledkům u obdobného experimentu u seznamu, akorát BVS není až tak jednoznačně nejlepší jako u seznamu. Tento jev bude pozorovatelný i u dalších experimentů s tabulkou.



Obrázek 7.1



Obrázek 7.2

7.2 Úspěšný Member

U úspěšného vyhledání jsme testovali čtyři různá pravděpodobnostní rozdělení – rovnoměrné, geometrické klasické, geometrické inverzní, geometrické náhodné.

7.2.1 Rovnoměrné rozdělení

(obr. 7.3, 7.4, 7.5)

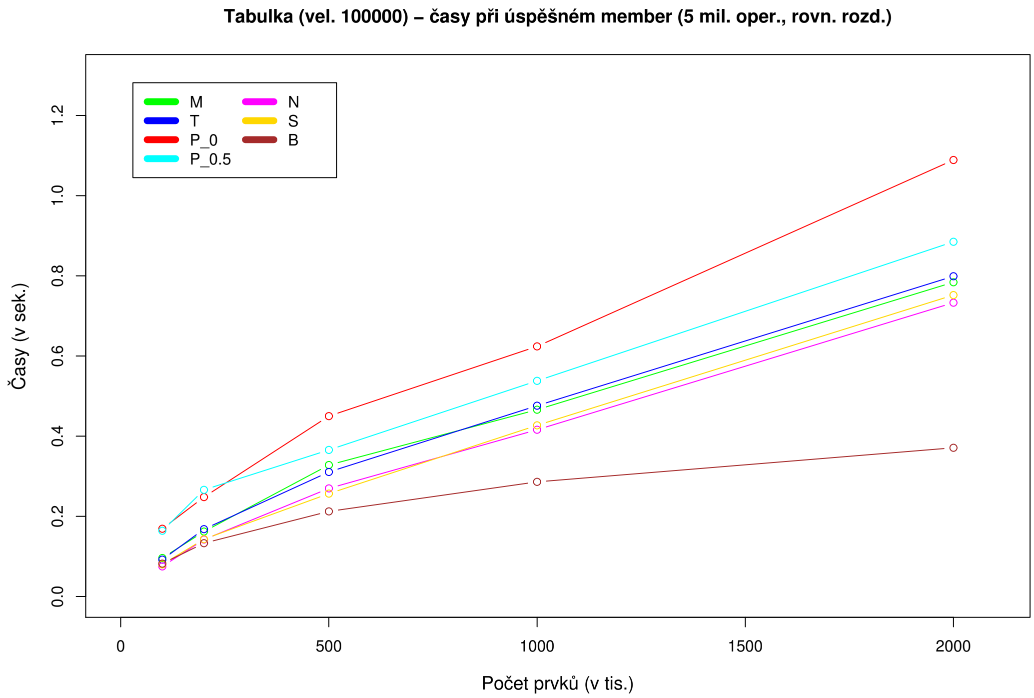
U rovnoměrného rozdělení je pro všechny strategie kromě BVS v podstatě stejný počet testů na vyhledání (7.2.1), proto se křivky překrývají. Počet všech testů pak mají TS(0) a TS(0.5) vyšší než počet testů na vyhledání, u ostatních strategií se u operace Member počet všech testů od počtu testů na vyhledání nikdy neliší. U BVS počet testů roste se zvyšujícím se faktorem naplnění pomaleji než lineárně, dá se předpokládat, že asi logaritmicky (alespoň pro samotný strom víme, že závislost počtu testů při úspěšném vyhledání na počtu prvků je logaritmická). Hlavně při vyšším zaplnění tabulky je jednoznačně lepší použití BVS než jiných strategií. Naměřené časy docela dobře odpovídají počtům všech testů, určitě lépe než jen počtům testů na vyhledání. Pokud to porovnáme s odpovídající situací u seznamů, tak hlavní rozdíl je podobně jako při vkládání v tom, že BVS už nemá takovou jasnou převahu nad ostatními strategiemi.

Pro klasickou strategii a strategii Sorted existuje společný teoretický odhad říkající, že očekávaný počet testů na vyhledání při rovnoměrném rozdělení se dá aproximovat jako $1 + \frac{\alpha}{2}$. To znamená pro 2 miliony prvků a tabulku velikosti 100 tisíc očekávaný počet testů 11. V experimentu se provedlo u klasické strategie i strategie Sorted pro 5 milionů operací skoro přesně 55 milionů testů, tedy výsledek je v souladu s teoretickým odhadem. Protože prakticky stejný počet testů na vyhledání byl naměřen i u ostatních seznamových strategií, dá se předpokládat, že pokud se úspěšně vyhledává dle rovnoměrného rozdělení, tak i pro ně platí odhad očekávaného počtu testů na vyhledání $1 + \frac{\alpha}{2}$.

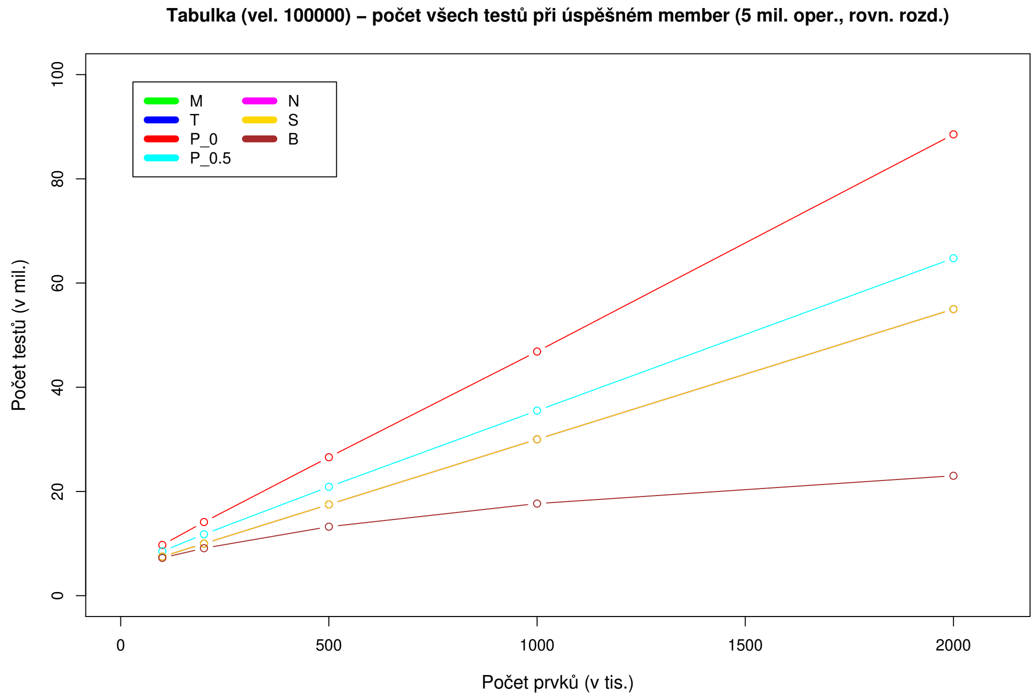
7.2.2 Geometrické rozdělení klasické

(obr. 7.6, 7.7, 7.8)

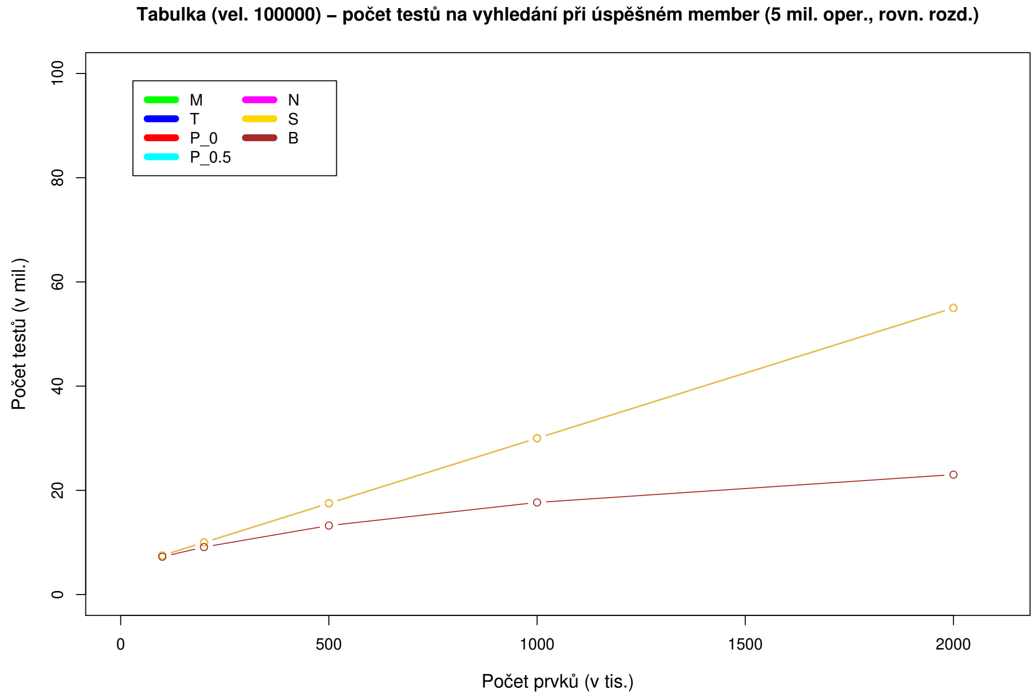
U tohoto rozdělení má logicky velmi nízký počet testů právě klasické řešení, protože má prvky seřazeny celou dobu v ideálním pořadí vzhledem k danému rozdělení. Na počty testů na vyhledání je o něco málo lepší BVS a o něco málo horší TS(0). Největší počet testů na vyhledání můžeme vidět u Sorted, pro kterou platí, že díky nezávislosti pořadí prvků na pořadí vkládání má prakticky stejné výsledky pro všechna testovaná rozdělení. Nejhorší výchozí pozici, co se týče pořadí prvků na začátku vyhledávání, má strategie MTF, protože prvky s největší pravděpodobností hledání jsou úplně vzadu, ale díky samoupravující technice jsou i zde výsledky relativně dobré a pro $\alpha \leq 10$ je počet všech testů dokonce lepší než u TS(0). Pro zvyšující se α je vidět velký nárůst počtu testů u TR. Časy opět docela odpovídají počtům všech testů.



Obrázek 7.3



Obrázek 7.4



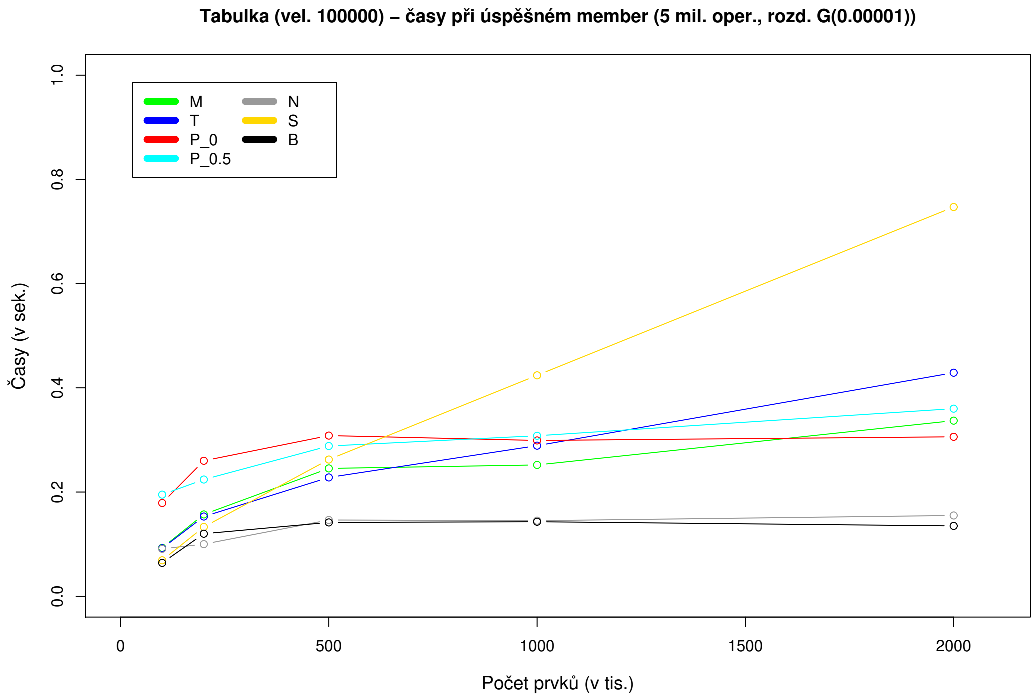
Obrázek 7.5

V grafu pro počet testů na vyhledání si můžeme všimnout, že počet testů je u některých strategií od 500 tisíc prvků dále už v podstatě stejný a u dalších nárůst také není nijak výrazný. Je to způsobeno tím, že k těm dalším prvkům už se stejně skoro nepřistupuje (při tomto rozdělení je 99,5 % hledání v prvních 500 tisících prvcích), a tak především u strategií, které mají na začátku hledání prvky seřazené podle pořadí vkládání, není skoro žádný rozdíl mezi počtem testů pro 500 tisíc prvků a pro 2 miliony prvků.

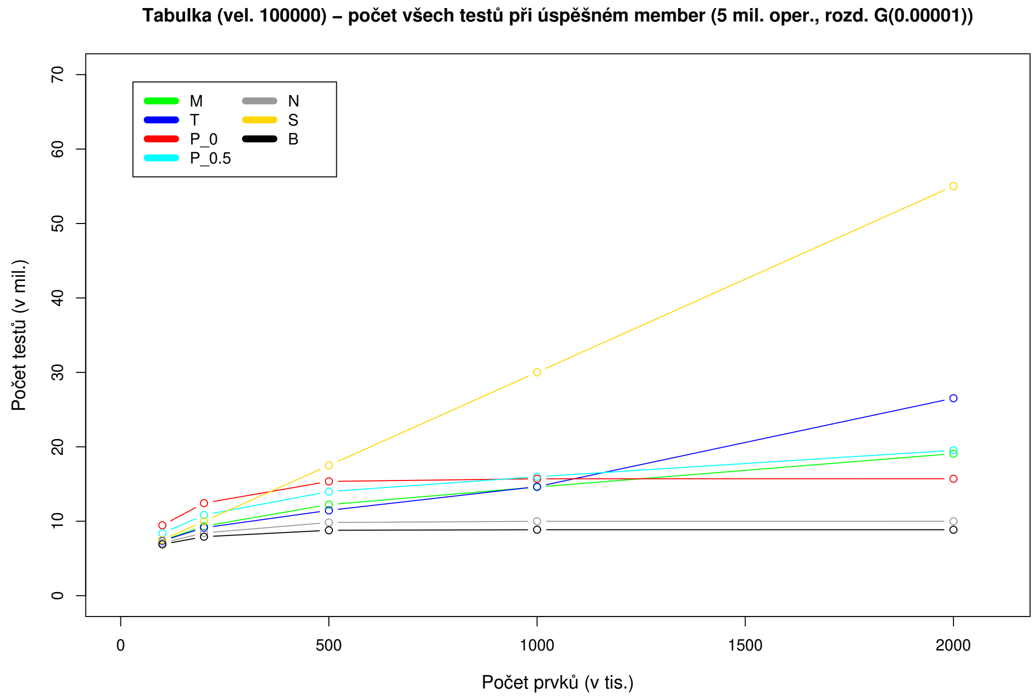
Pokud porovnáme výsledky s odpovídajícím experimentem u seznamů, tak hlavní rozdíl je kromě BVS ještě u strategie TR. Ta má totiž zde na rozdíl od experimentů se seznamy dost odlišné výsledky (jak počty testů, tak časy) od klasické strategie. Je to dáno tím, že zatímco u samotného seznamu se pořadí prvků na začátku hledání liší u TR od klasické strategie (která má ideální pořadí vzhledem k danému rozdělení) pouze v jediném prvku – tom prvně vkládaném, tak u tabulky je tento rozdíl u všech prvně vkládaných prvků do jednotlivých seznamů, které k tabulce patří, tedy např. v tomto experimentu takových je až 100 tisíc. Tyto prvky jsou na začátku hledání na konci jednotlivých seznamů, a protože při relativně rovnoměrném zaplňování tabulky se dá počítat s tím, že všechny byly vkládané ze začátku, tak mají i vysokou pravděpodobnost vyhledávání, a proto jejich počáteční umístění vzadu velice zvýší celkový počet testů.

7.2.3 Geometrické rozdělení inverzní

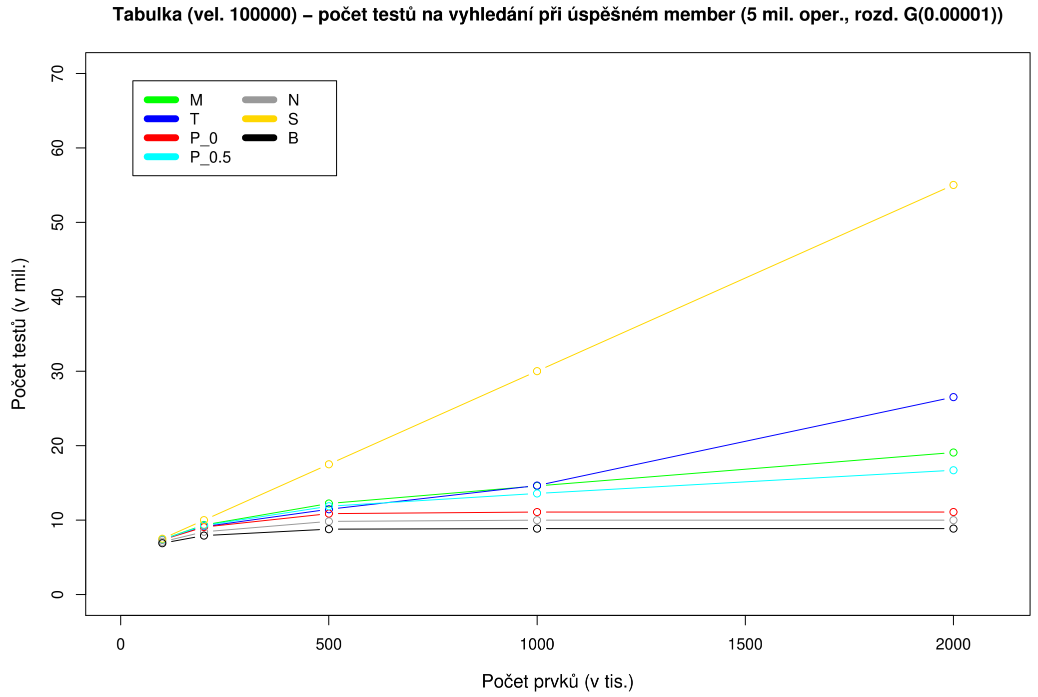
(obr. 7.9, 7.10, 7.11)



Obrázek 7.6



Obrázek 7.7



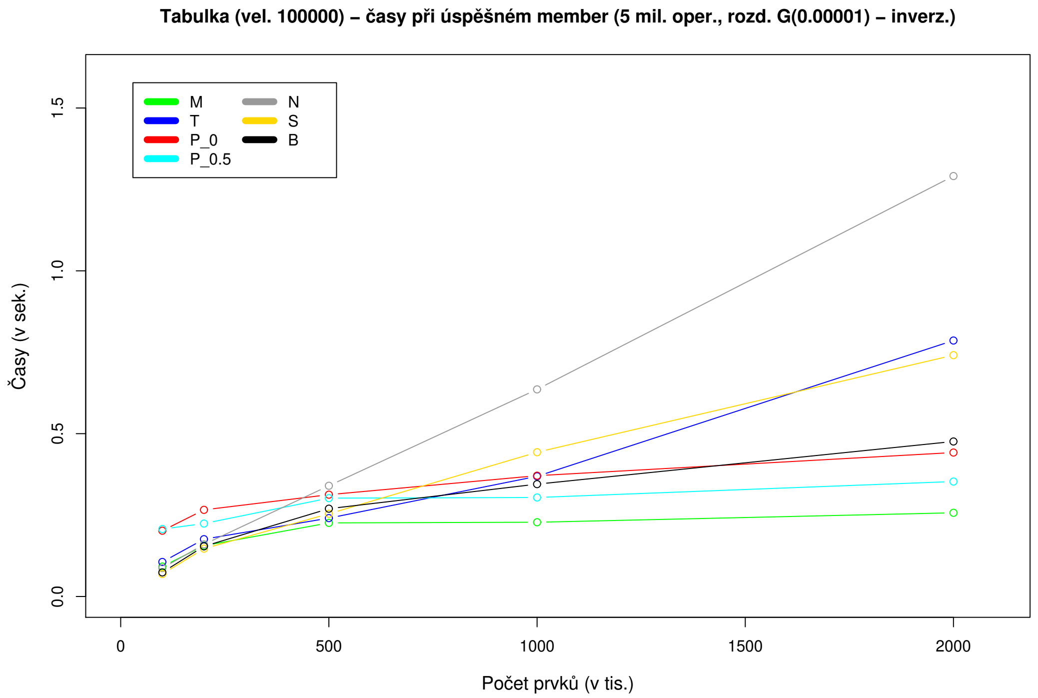
Obrázek 7.8

Při použití tohoto rozdělení platí, že pravděpodobnost vyhledání prvků je tím větší, čím později byl vložen. Celkem tedy nepřekvapí, že nejlepších výsledků dosahuje strategie MTF. Co se týče počtu testů na vyhledání, drží s ní krok ještě TS(0), TS(0.5) a při menším zaplnění tabulky také TR. Naopak klasická strategie vychází dost špatně, protože prvky, které se vyhledávají nejčastěji, se nepřesouvají do „výhodnějších" pozic, ale zůstávají na konci v seznamu. U strategie TR je opět vidět špatné chování na delších řetězcích. To je způsobené tím, že s větší délkou řetězců se ty nejvyhledávanější prvky vyskytují na začátku vyhledávání čím dál tím dál od začátku řetězce. Ve srovnání s obdobným experimentem na seznamu má strategie TR nižší počty testů než klasická strategie, což je způsobeno tím, že zde je počet operací pro daný počet prvků dostačující na to, aby se projevila výhoda samoupravující strategie.

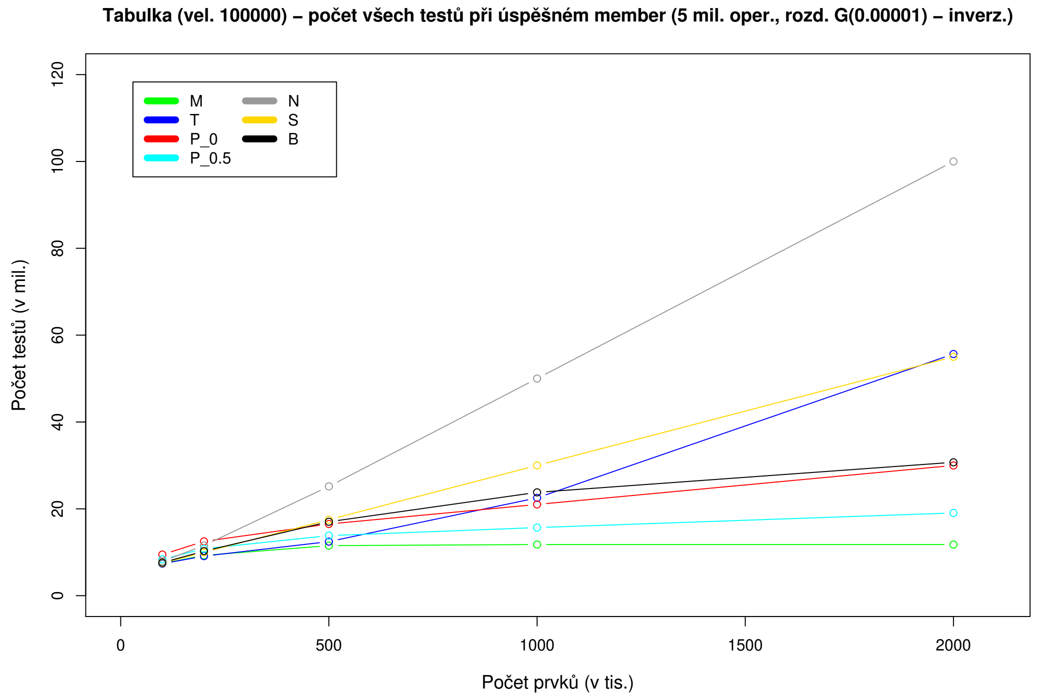
7.2.4 Geometrické rozdělení náhodné

(obr. 7.12, 7.13, 7.14)

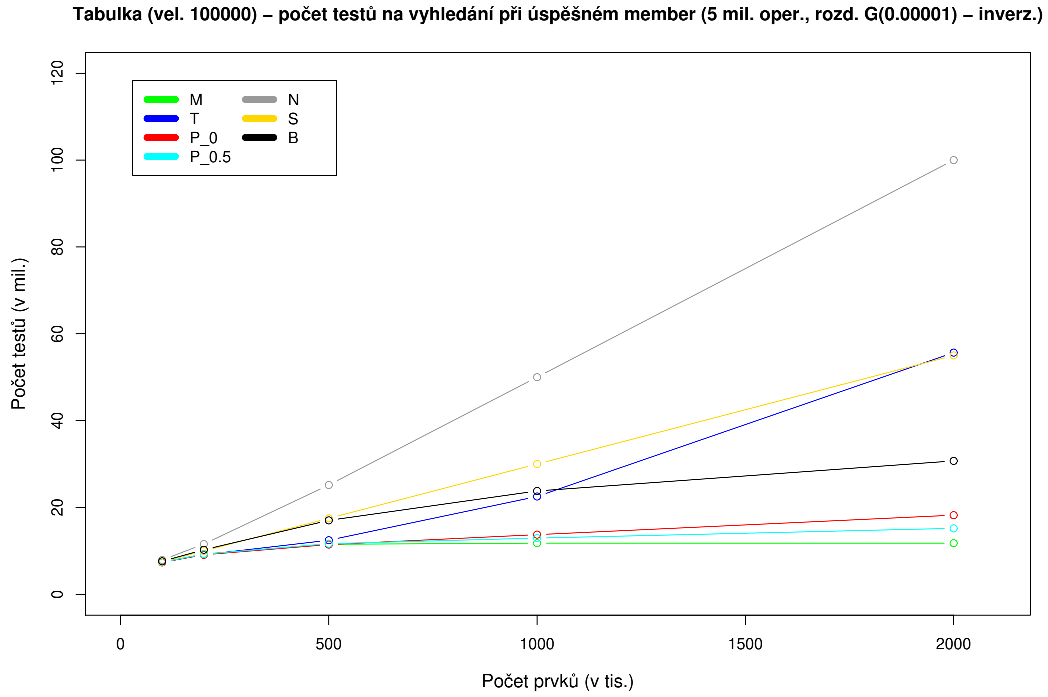
Při tomto typu geometrického rozdělení není pravděpodobnost vyhledání prvků nijak vázána na pořadí vkládání prvků. Počty testů u klasické strategie se shodují s počty testů u strategie Sorted (protože setřídění pouze způsobí další náhodné promíchání). V počtech testů na vyhledání je tentokrát nejlepší TS(0), ale MTF a TS(0.5) nezůstávají o moc pozadu. V počtech všech testů už je nej-



Obrázek 7.9



Obrázek 7.10



Obrázek 7.11

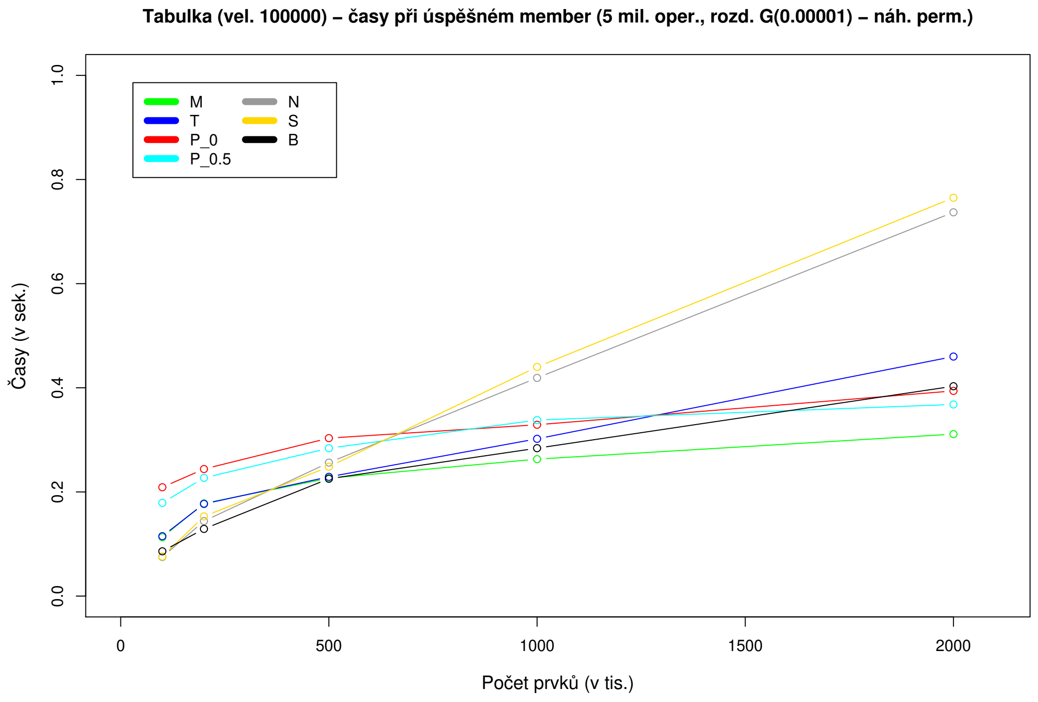
lepší MTF (pro nízké zaplnění tabulky možná trošičku překonaná strategiemi BVS a TR), zato TS(0) a TS(0.5) jsou v počtech všech testů pro nižší zaplnění tabulky dokonce horší než klasická strategie. Metoda TR se pro kratší řetězce chová stejně dobře jako MTF, ale pro delší řetězce už dochází k výraznému zhoršení. Zatímco u vyšších faktorů naplnění jsou jednoznačně výhodnější samoupravující strategie, tak pro nízké α (1 nebo 2) už mají nesamoupravující strategie v podstatě stejný počet testů. A časy mají dokonce nižší, protože se nedělá žádná úprava seznamu (stromu).

Ve srovnání s analogickým experimentem na seznamu má strategie TR nižší počty testů než klasická strategie a platí zde stejné zdůvodnění jako u geometrického rozdělení klasického.

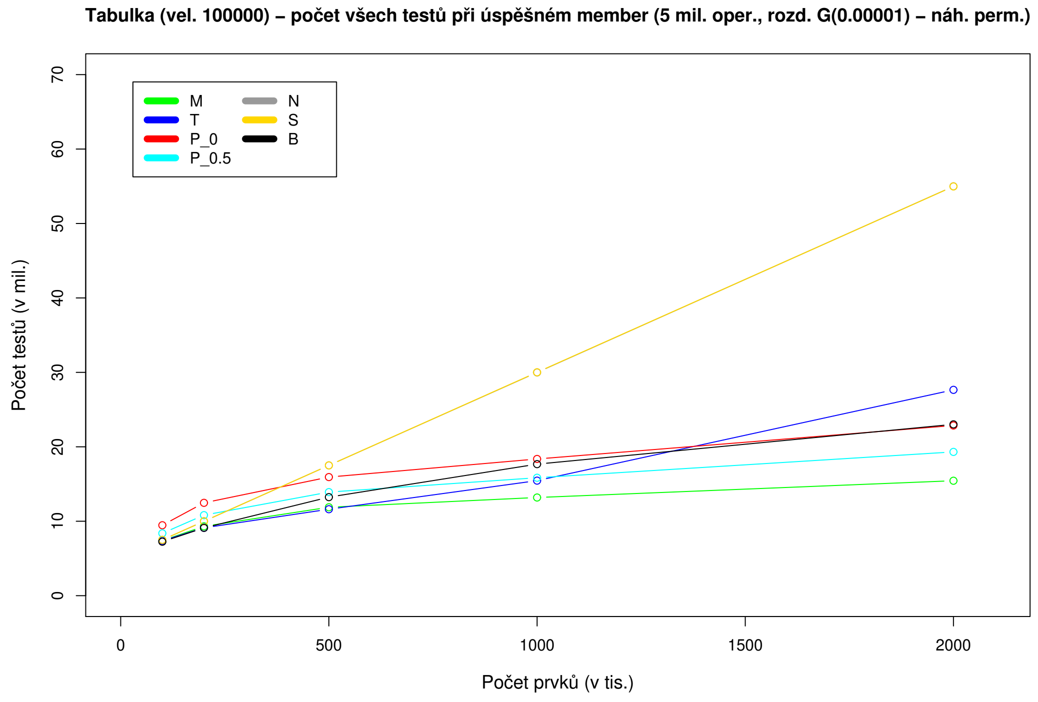
7.2.5 Porovnání s OSO

Nyní se budeme zabývat poměry jednotlivých strategií vůči OSO (co to přesně znamená, je vysvětleno v 6.2.5). Následující graf (7.15) zobrazuje tyto poměry pro tabulku velikosti 100 tisíc s 500 tisíci prvky v závislosti na počtu operací. Pro lepší přehlednost jsme na ose x použili logaritmickou stupnici.

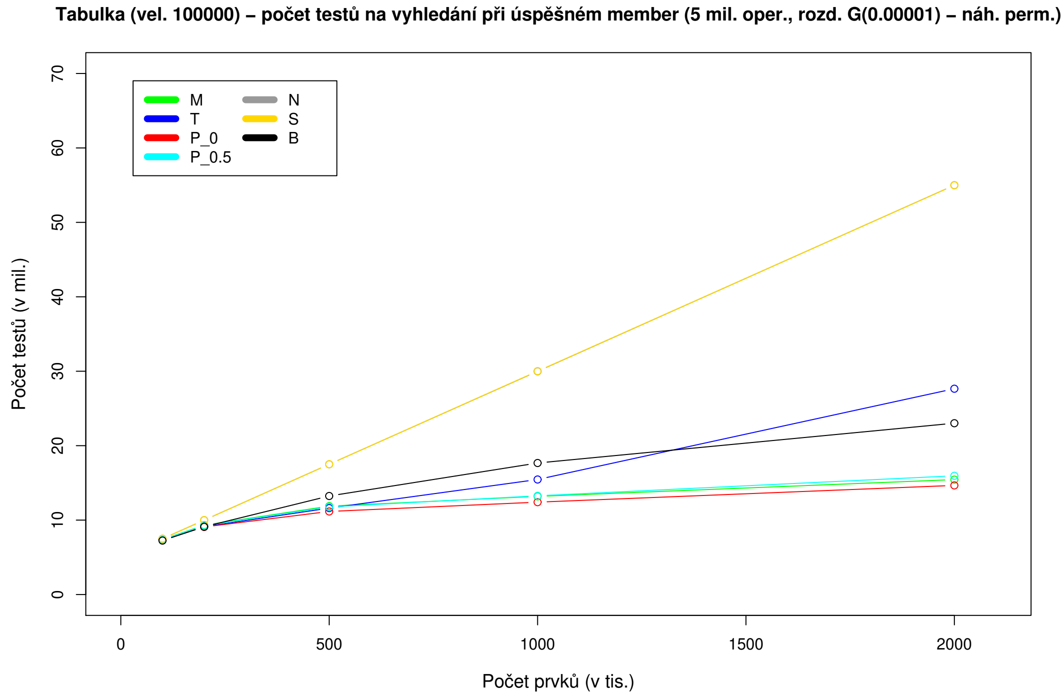
U nesamoupravujících strategií se poměr vůči OSO s rostoucím počtem operací nemění. U klasické strategie a Sorted zůstává kolem hodnoty 1.78, u BVS asi 1.35. U samoupravujících strategií dochází k poklesu poměru vůči OSO, především během prvního milionu operací. Ze začátku jsou na tom všechny samoupravující strategie o dost hůře než BVS, ale po dvou milionech operacích už jsou



Obrázek 7.12



Obrázek 7.13

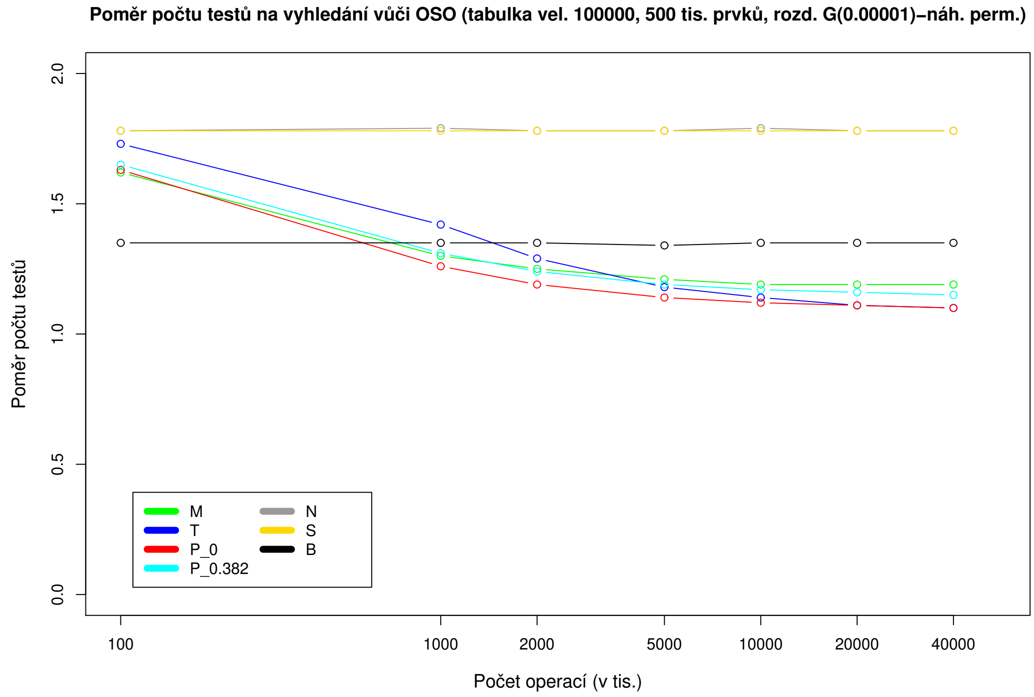


Obrázek 7.14

všechny lepší než BVS.

U seznamu je dokázáno, že TR má nižší asymptotickou očekávanou cenu než MTF, ale konverguje k ní pomaleji. Toto chování můžeme pozorovat i v případě tabulky. Pro prvních několik milionu operací je TR ze samoupravujících strategií nejhorší, ale nakonec se dostane až na první místo společně s TS(0). Rychlost konvergence je také závislá na délce seznamu v tabulce. Pokud se podíváme na obr. 7.14, uvidíme, že zatímco pro 500 tisíc prvků už 5 milionů operaci stačí na to, aby TR byla lepší než MTF, tak pro 2 miliony prvků je TR výrazně horší než MTF. Aby se dostala na úroveň MTF, tak by v tomto případě potřebovala ještě o dost větší počet operací. Při zvýšení počtu prvků sice i MTF zřejmě konverguje pomaleji, ale u TR se to projevuje daleko výrazněji.

Pro samotný seznam existují odhady pro poměr asymptotické očekávané ceny určité samoupravující strategie a asymptotické očekávané ceny OSO. Pro tabulku nějaké podobné odhady zřejmě nejsou, přesto ale uvedeme aspoň poměry získané v našich experimentech a porovnáme je mezi sebou. Vezmeme poměry pro největší testovaný počet operací, kdy už se poměry moc nemění a zde získané hodnoty už mají asymptotický charakter (velice se přibližují hodnotám, které bychom dostali pro nekonečný počet operací). Poměr u strategie MTF je 1.19, u TS(0.5) 1.16 a u TS(0) a TR 1.11. Tyto poměry jsou nižší než u seznamu, což je vcelku logické, protože v krátkých seznamech tabulky se nemůže projevit síla OSO tak dobře jako v jednom dlouhém seznamu (u prvků z různých seznamů OSO sice „ví“, který bude vyhledávanější, ale nelze to nijak využít). Pro příklad si můžeme představit tabulku s jednoprvkovými seznamy, tam by poměry pro všechny strategie byly 1.



Obrázek 7.15

7.2.6 Zhodnocení

Pro rovnoměrné rozdělení je nejvýhodnější strategie BVS, u geometrického rozdělení klasického je nejlepší BVS nebo klasická strategie, a u zbylých dvou pro nízký faktor zaplnění ($\alpha \leq 2$) klasická strategie nebo BVS a pro vyšší faktor zaplnění MTF. Na rozdíl od experimentů se seznamy už tedy není BVS nejlepší vždy a za všech okolností.

Zajímavé chování se vyskytuje u strategie TR, u které při nerovnoměrném rozdělení s rostoucím faktorem naplnění výrazně narůstá počet testů (a zároveň s tím i časy). Je to způsobeno tím, že při delších řetězcích by bylo třeba více operací, aby TR „zkonvergovala“ (tedy aby se její cena přiblížila té asymptotické). U geometrického rozdělení náhodného tedy platí, že čím více máme v tabulce prvků (velikost tabulky a počet operací zůstávají), tím více se počet testů u TR blíží počtu testů u klasické strategie (měřeno poměrem), která představuje stav, než proces konvergence začal.

Nesamoupravující strategie mají pro rovnoměrné a geometrické náhodné stejné výsledky. Strategie Sorted má dokonce stejné výsledky i pro všechny typy geometrického rozdělení, protože pořadí prvků u ní není závislé na pořadí vkládání.

Celkově lze říci, že pokud bychom toho o datech moc nevěděli a měli bychom vybrat co nejlepší strategii pro praktické použití, tak by to byla nejspíš BVS, dalším kandidátem by bylo MTF, které by bylo asi nejlepším kandidátem v případě nerovnoměrného rozdělení. Pokud nebudeme mít tabulku příliš zaplněnou, tak je k MTF dobrou alternativou také TR. Při větším zaplnění TR potřebuje velké množství operací, aby se začala chovat rychle. Konkrétní hodnota faktoru zaplnění, do kterého se TR chová srovnatelně s MTF je závislá na velikosti tabulky, podobě série požadavků (distribuci), počtu operací. V našem případě (velikost tabulky 100 tisíc, počet operací 5 mil., geom. rozd. s koef. 0.00001) tou hranicí bylo přibližně $\alpha = 5$.

7.3 Neúspěšný Member

(obr. 7.16, 7.17)

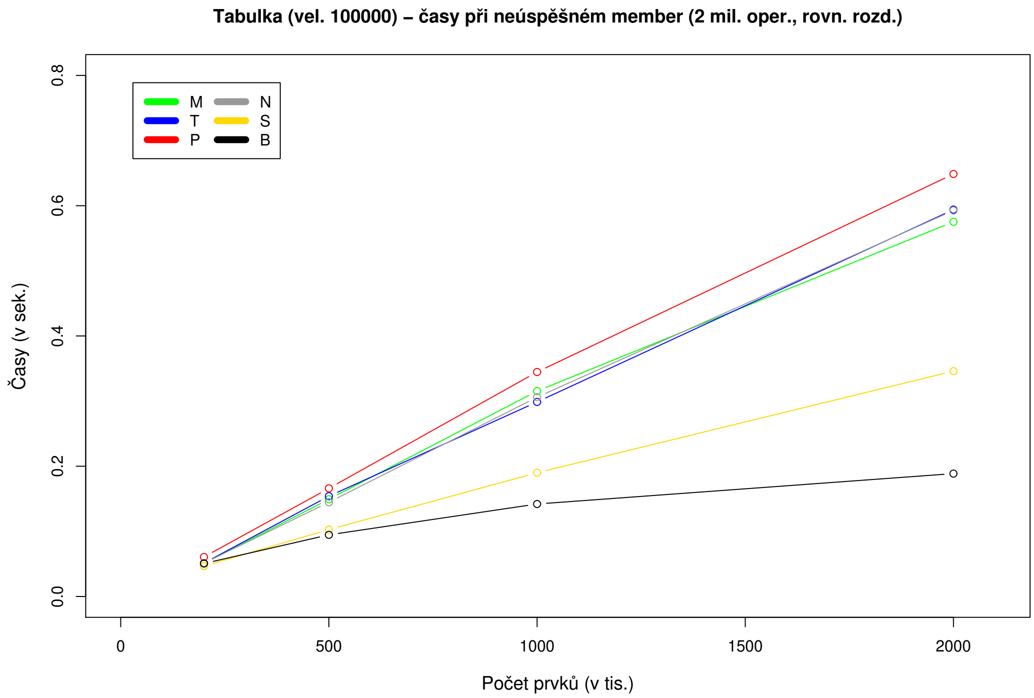
Jak už bylo řečeno u seznamů, při neúspěšném vyhledávání se samoupravující strategie chovají úplně stejně jako klasická strategie. U algoritmu TS(p) nezáleží vůbec při neúspěšném vyhledání na hodnotě parametru p, proto jsme provedli experiment pro jednu libovolnou a do grafu její hodnotu ani nepsali.

Výsledky u neúspěšného vyhledání jsou podobné těm u vkládání, což je i logické, protože úspěšnému vkládání (kterých je u testování operace Insert naprostá většina) vždy předchází neúspěšné vyhledání a pak už je navíc pouze vložení nového prvku. Podobně jako u vkládání je tedy nejvýhodnější strategií BVS, na druhém místě je Sorted a na třetím místě je skupina zbylých strategií, které mají všechny počty testů úplně stejné a časy přibližně také. Výsledky se také vcelku shodují s výsledky testování u samotného seznamu.

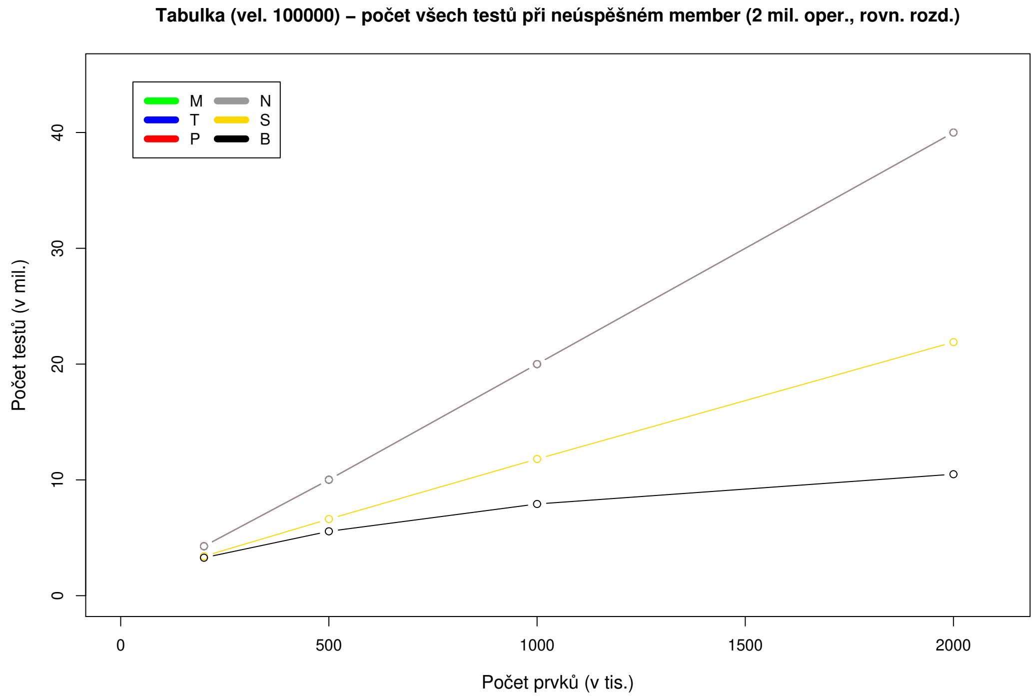
Pro klasickou strategii a strategii Sorted existují odhady pro počty testů, které říkají, že očekávaný počet testů při neúspěšném vyhledání je u klasické strategie $e^{-\alpha}+\alpha$ a u strategie Sorted $e^{-\alpha}+1+\frac{\alpha}{2}-\frac{1}{\alpha}(1-e^{-\alpha})$. Pro 2 miliony prvků a velikost tabulky 100 tisíc dávají tyto odhady hodnoty 20 pro klasickou strategii a 10.95 pro Sorted (ty hodnoty jsou v podstatě přesné, e^{-20} je zanedbatelné). V experimentu jsme pro 2 miliony operaci provedli u klasické strategie skoro přesně 40 milionů testů na vyhledání u klasické strategie a 21.9 milionů testů u strategie Sorted, tedy výsledky odpovídají teoretickým odhadům.

Poznámka

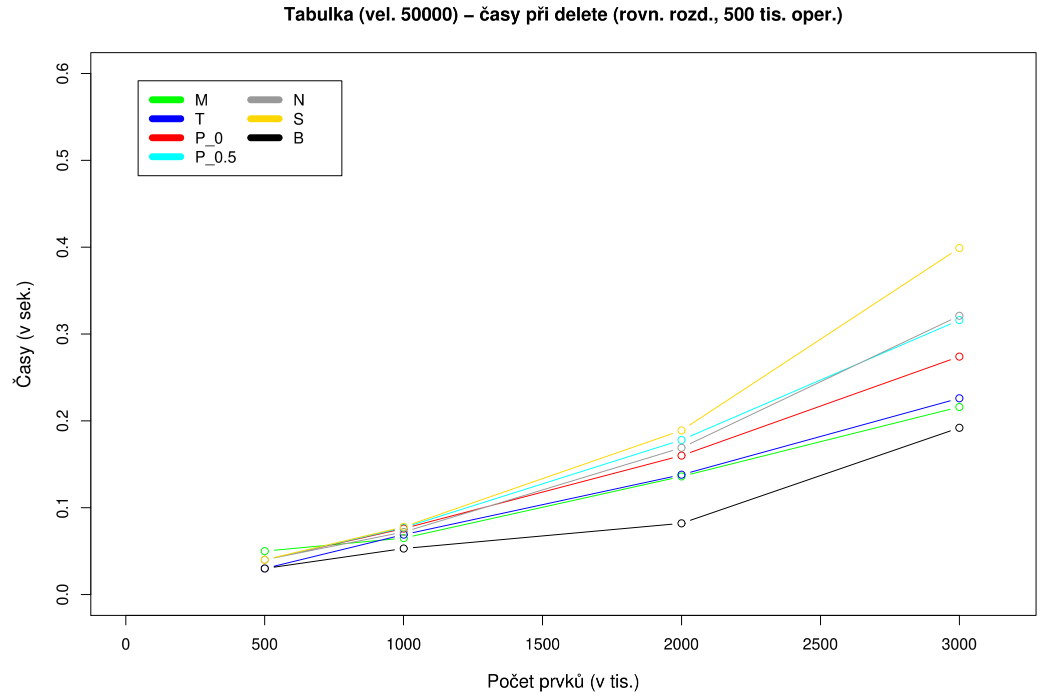
Neuvádím graf s počtem testů na vyhledání, protože ten je stejný jako graf s počtem všech testů. Žádné testy po vyhledání se totiž u neúspěšného Member nevyskytují.



Obrázek 7.16



Obrázek 7.17



Obrázek 7.18

7.4 Delete

V této podkapitole jsou uvedeny výsledky testů pro operaci Delete na tabulkách. Podobně jako u seznamu i zde budou na počty testů vycházet prakticky stejně dvě dvojice strategií, jednak klasická strategie a TS(0) a potom Sorted a TS(0.5). Proto překrývání jejich křivek už nebudeme u jednotlivých grafů komentovat.

Platí zde stejná poznámka, která byla uvedená už u seznamů (viz 6.4), zdůvodňující, proč neuvádím grafy s testy na vyhledání a grafy pro geometrické rozdělení náhodné.

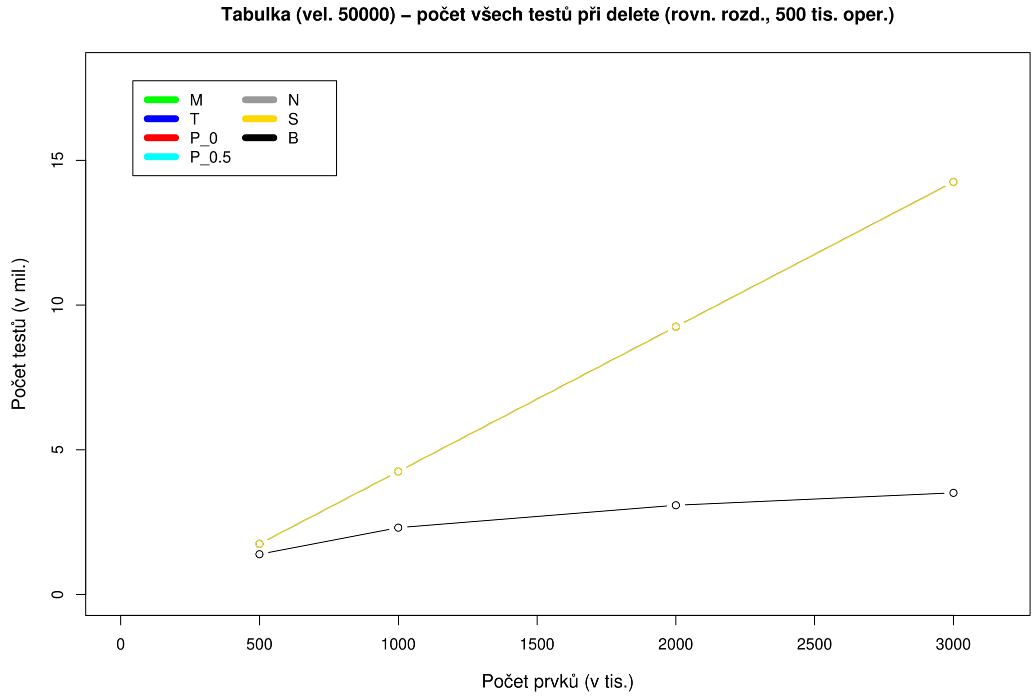
7.4.1 Rovnoměrné rozdělení

(obr. 7.18, 7.19)

V grafu počtu testů máme jen dvě křivky, jednu pro strategii BVS a druhou společnou pro všechny ostatní strategie. Přestože po vložení prvků už je u jednotlivých strategií se seznamy různé pořadí prvků, způsobí rovnoměrné rozdělení nakonec vyrovnanost počtu testů.

7.4.2 Geometrické rozdělení klasické

(obr. 7.20, 7.21)



Obrázek 7.19

Při tomto rozdělení se mažou ze začátku spíš prvky, které byly vkládané dřív, a ty vkládané později se mažou většinou až později. A protože z hlediska počtu testů je výhodnější nejprve mazat prvky ze začátku seznamu, tak logicky nejmenší počty testů mají ze seznamových strategii klasická strategie a TS(0), které mají prvky seřazené přesně tak, jak se vkládaly, a naopak nejhůře dopadá MTF.

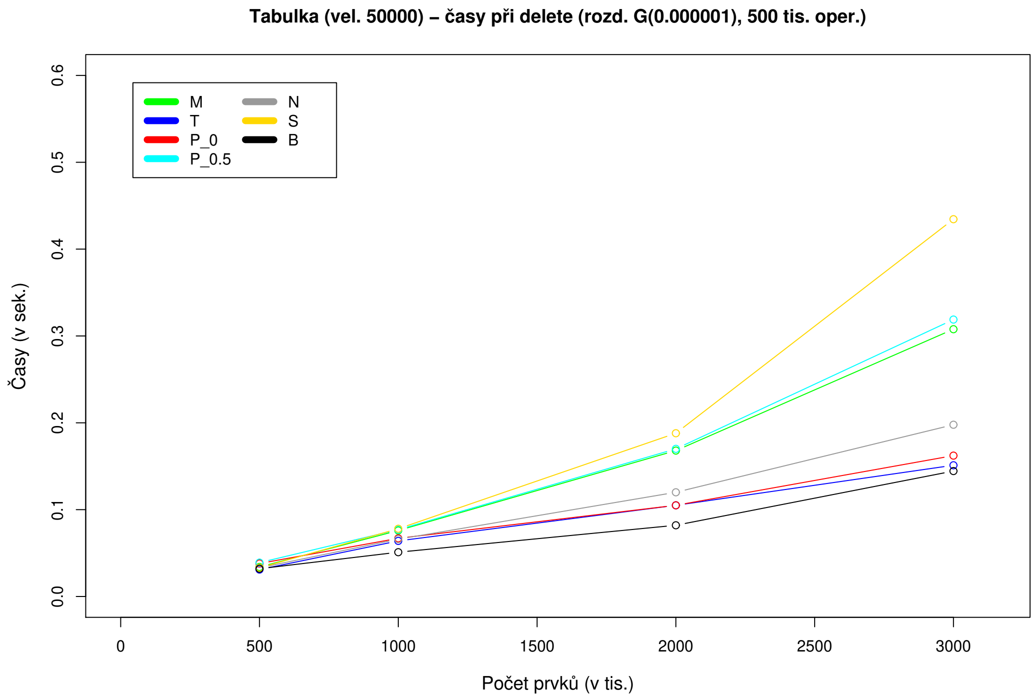
7.4.3 Geometrické rozdělení inverzní

(obr. 7.22, 7.23)

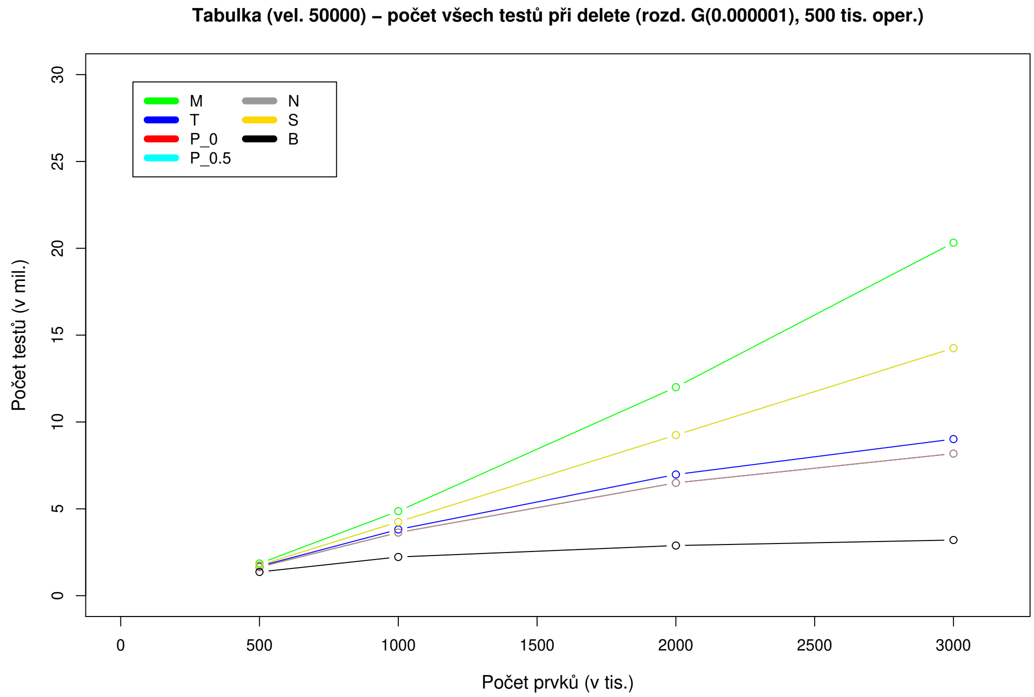
Při tomto rozdělení dochází k opačné situaci než v předchozím případě. Je větší šance, že se ze začátku budou mazat prvky vkládané později než prvky vkládané dříve. Proto je ze seznamových strategii nejrychlejší MTF a naopak nejpomalejší jsou klasická strategie a TS(0) (jejich křivky pro počet testů se překrývají). Úplně nejlepší je opět BVS a to i přesto, že toto rozdělení je pro něj z těch testovaných nejhorší, protože při něm ty nejvyhledávanější prvky jsou nejhlouběji ve stromě.

7.4.4 Zhodnocení

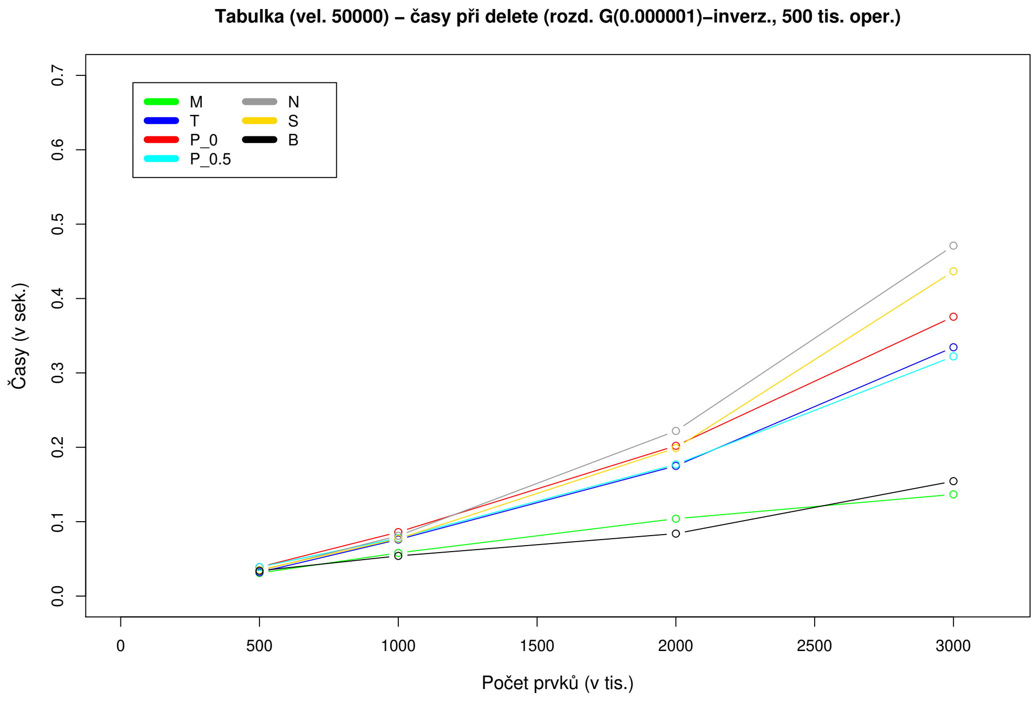
Nejvýhodnější strategií u operace Delete je BVS, ostatní strategie vycházejí v průměru přibližně nastejno. Z hlediska samoupravujících strategií operace Delete není zajímavá, protože prvek se po operaci v seznamu už dále nevyskytuje. Výsledky se velice shodují s výsledky experimentů na seznamech. Můžeme si ale všimnout



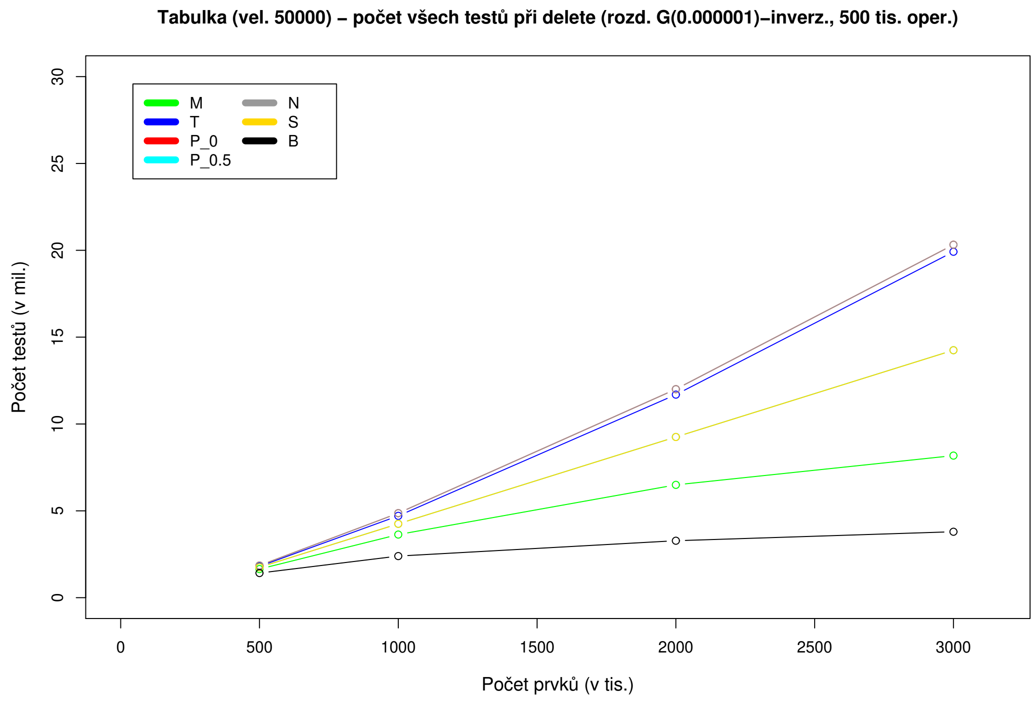
Obrázek 7.20



Obrázek 7.21



Obrázek 7.22



Obrázek 7.23

mírného rozdílu mezi počty testů u klasické strategie a strategie TR, zatímco u seznamů se v podstatě shodovaly. To je způsobeno tím, že u seznamu se pořadí prvků po vložení liší pouze tím, že první vložený prvek je na konci, což počet testů nijak zvlášť neovlivní, ale u tabulky se tato situace objeví v každém neprázdném řetězci (lépe řečeno nějakou roli to hraje až u řetězců délky alespoň dva), a to už počet testů ovlivní nezanedbatelným způsobem.

7.5 Kombinace

V této podkapitole jsou uvedeny výsledky experimentů pro kombinaci operací Insert, úspěšný Member, neúspěšný Member a Delete. Byla použita stejná procentuální zastoupení jednotlivých operací jako u experimentů na seznamu – Insert 16 %, úspěšný Member 62 %, neúspěšný Member 16 %, Delete 6 %. U Insert a neúspěšného Member bylo rozdělení vždy rovnoměrné, u úspěšného Member a Delete bylo vyzkoušeno i nerovnoměrné.

7.5.1 Úspěšný Member – rovnoměrné, Delete – rovnoměrné

(obr. 7.24, 7.25, 7.26)

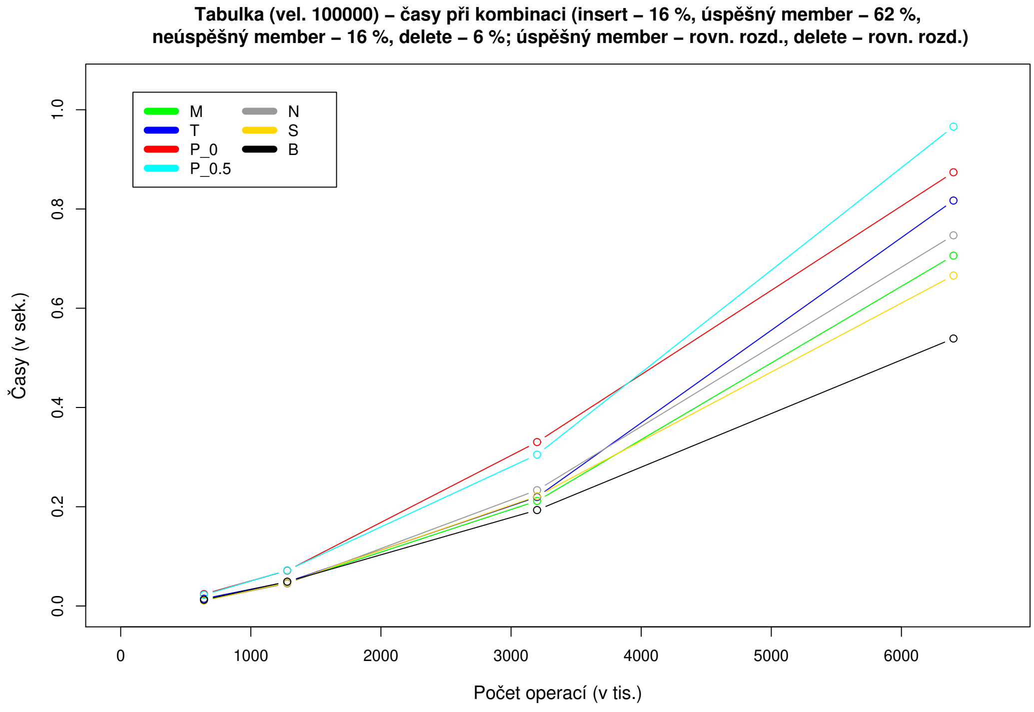
Počet testů na vyhledání je nejnižší u strategie BVS, ostatní strategie mají v podstatě (navzájem) stejný počet testů na vyhledání s výjimkou Sorted, která ho má trochu nižší, což je způsobeno menším počtem testů u Insert a neúspěšného Member.

Počty všech testů se mohou lišit od počtu testů na vyhledání pouze u TS(p) a BVS. U BVS je tento rozdíl zanedbatelný (nárůst méně než 1 %) a podobně tomu bude i u dalších rozdělení. Zato u TS(0.5) a především u TS(0) už je počet všech testů o dost vyšší než počet testů na vyhledání. Časy odpovídají docela dobře počtům všech testů (jen pro max. testovaný počet operací vyšel čas u TS(0.5) větší než u TS(0)).

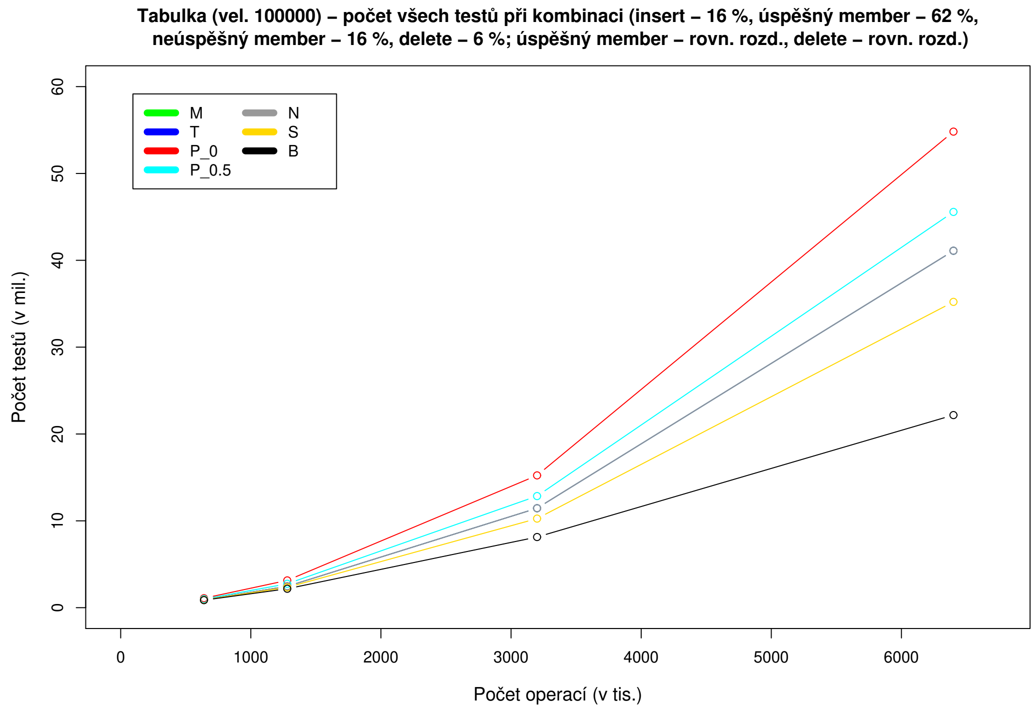
7.5.2 Úspěšný Member – geometrické klasické, Delete – rovnoměrné

(obr. 7.27, 7.28, 7.29)

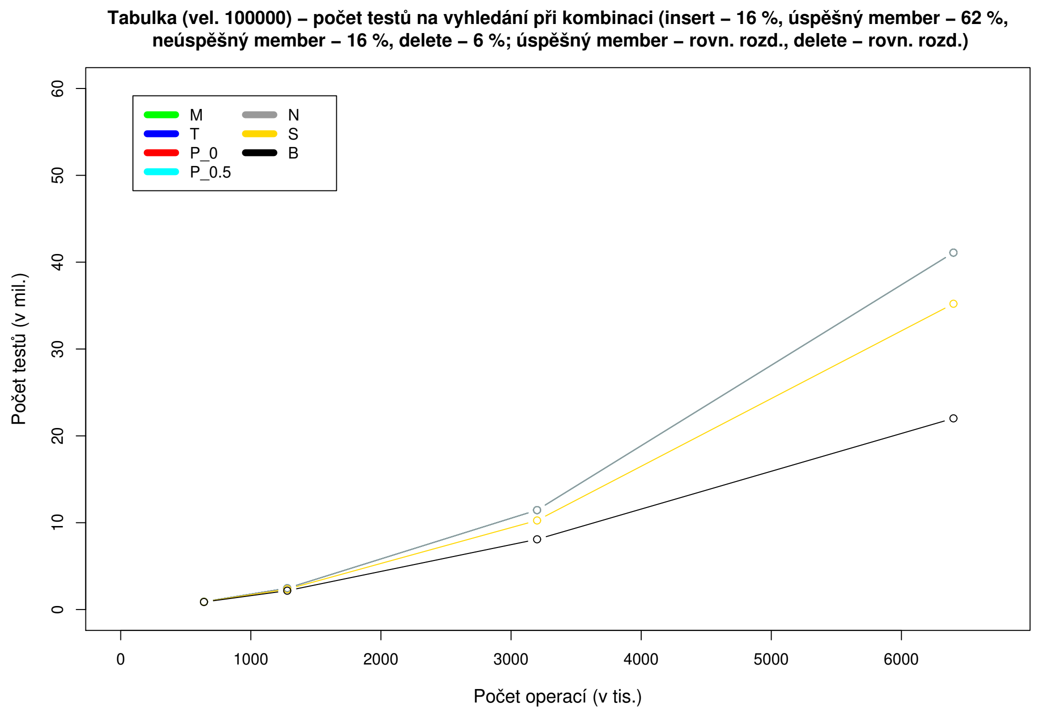
U operace úspěšný Member bylo nyní použito geometrické rozdělení klasické, díky čemuž je po BVS nejlepší strategií klasická strategie. V počtech testů na vyhledání za ní následuje těsně TS(0), pak další strategie a nejhorší je Sorted. V počtech všech testů je už Sorted nejhorší jen pro největší testovaný počet operací, pro nižší je lepší než TS(0) a TS(0.5). Stejný jev je vidět i u naměřených časů, které celkově docela dobře korespondují s počty všech testů. Ve srovnání s analogickým experimentem na seznamu jsou hlavní rozdíly u BVS, která už není tak suverénně nejlepší, a u TR, která zde má odlišné výsledky od klasické



Obrázek 7.24



Obrázek 7.25



Obrázek 7.26

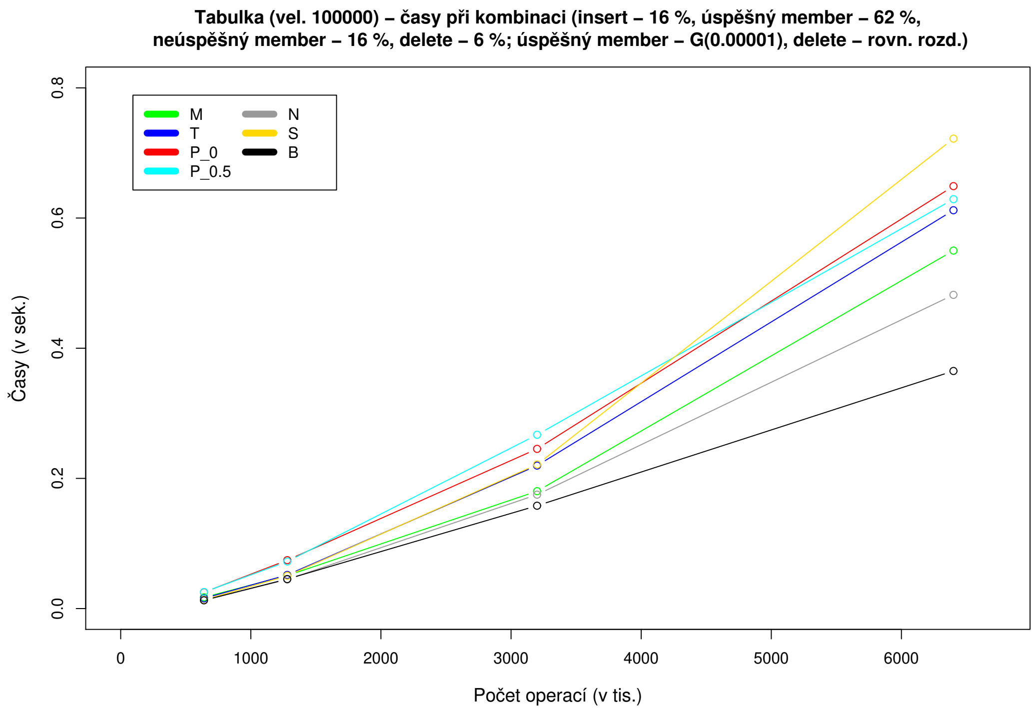
strategie (tento jev popsán více u úspěšného Member – 7.2.6). Ke grafům bych ještě podotkl, že v grafu pro počet testů na vyhledání se dost překrývají křivky pro TR a MTF, v grafu pro počet všech testů se pak ještě navíc částečně překrývají křivky pro TS(0) a TS(0.5).

7.5.3 Úspěšný Member – geometrické inverzní, Delete – geometrické klasické

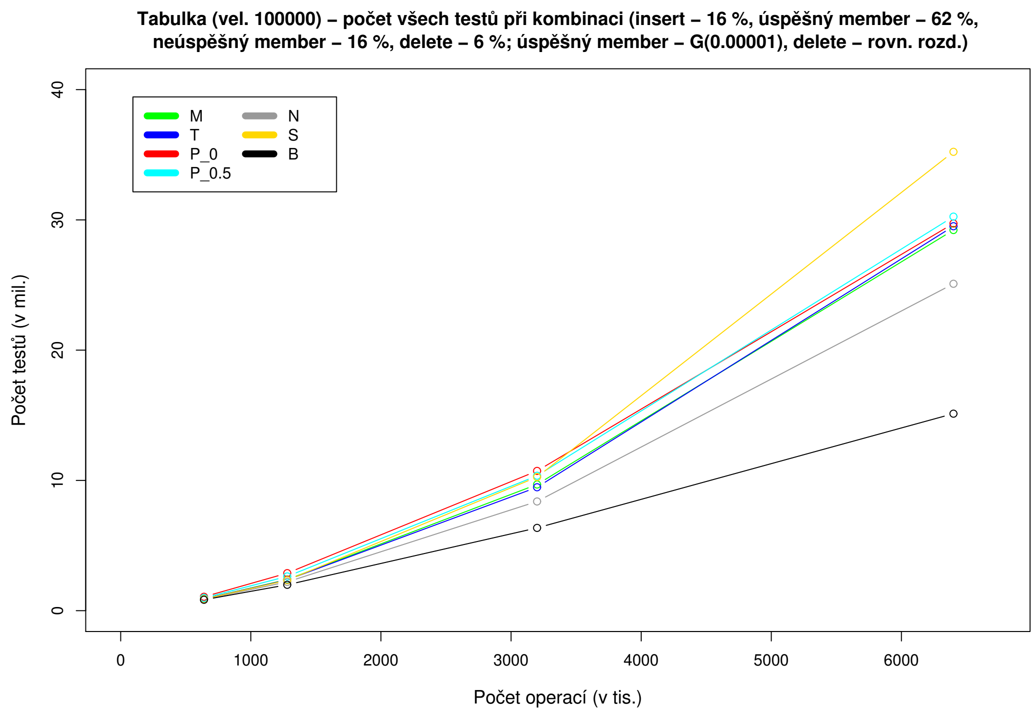
(obr. 7.30, 7.31, 7.32)

Pokud se podíváme na počty testů na vyhledání, tak jednoznačně nejhorší strategií je klasická strategie, naopak nejlepší (alespoň pro maximální počet testovaných operací) je BVS, těsně následovaná MTF, TS(0.5) a TS(0). V počtech všech testů dojde oproti počtu testů na vyhledání opět k nárůstu u TS(0) a TS(0.5) a nepatrně i u BVS (je to vidět porovnáním s MTF). Kvůli tomuto nárůstu už je na počet všech testů TS(0) horší než Sorted. U TR lze pozorovat jev, který byl vidět i u samotné operace úspěšný Member, totiž že pokud při použití nerovnoměrného rozdělení TR nemá k dispozici dostatečný počet operací, tak se pro vyšší faktory zaplnění nechová tak dobře jako ostatní samoupravující strategie. V naměřených časech jsou tentokrát vidět určitě rozdíly oproti počtům všech testů (např. ve vztahu MTF a BVS), ale ty hlavní tendence jsou stejné.

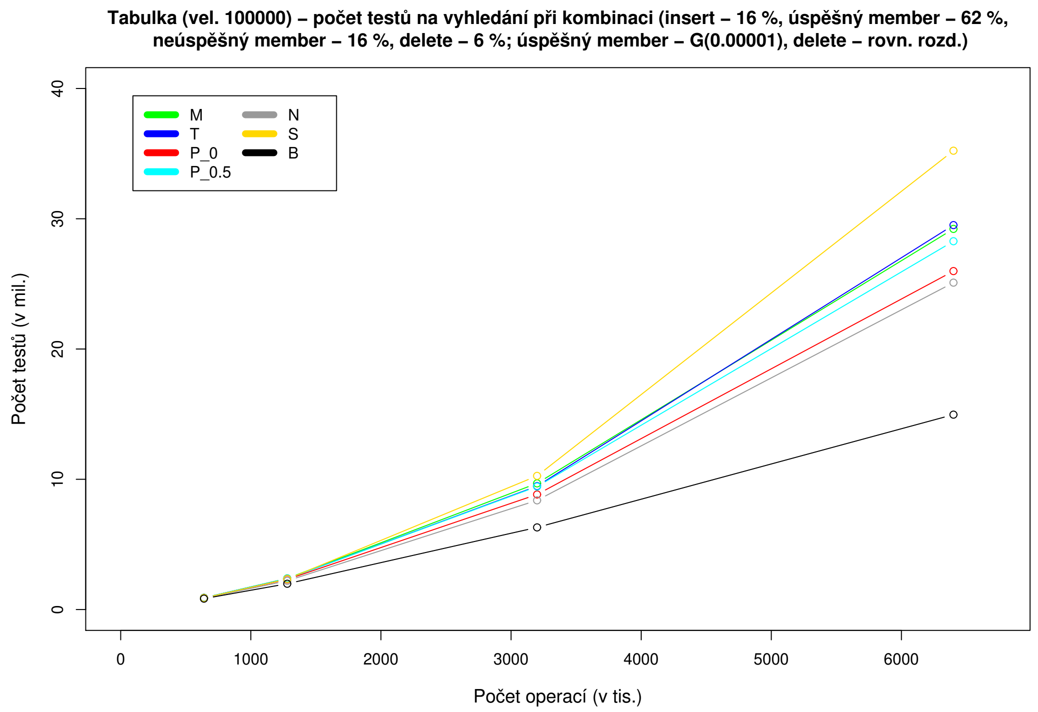
Podobně jako u seznamu i zde jsme dělali také experiment, který se od tohoto lišil pouze použitím rovnoměrného rozdělení pro Delete, a stejně jako u seznamu



Obrázek 7.27



Obrázek 7.28



Obrázek 7.29

byly opět výsledky v podstatě totožné.

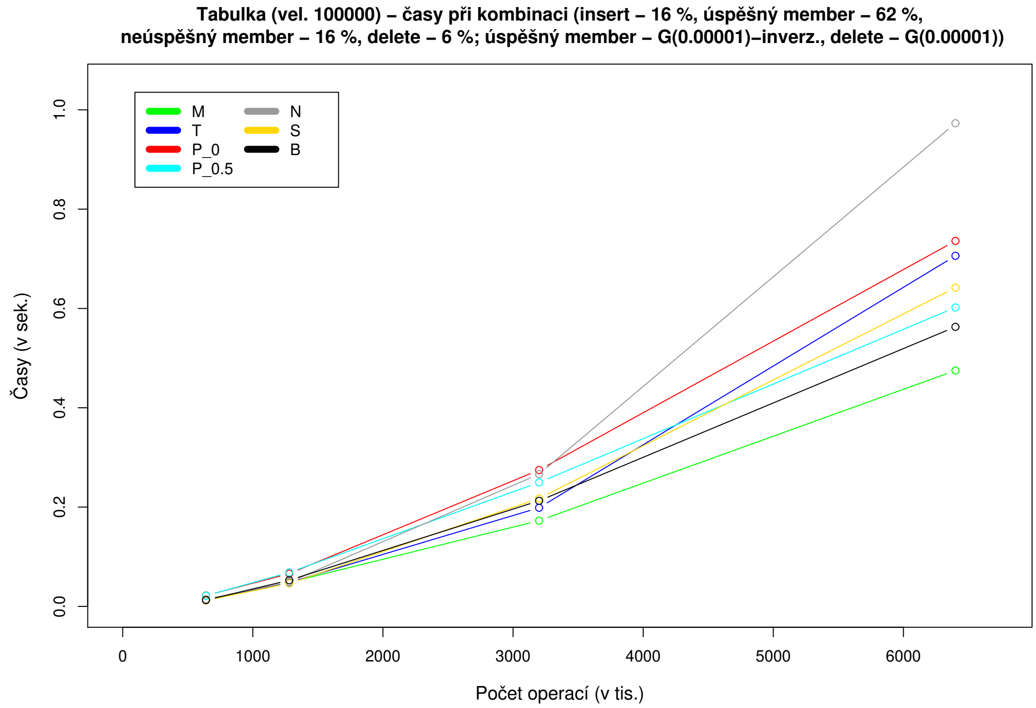
7.5.4 Úspěšný Member – geometrické náhodné, Delete – rovnoměrné

(obr. 7.33, 7.34, 7.35)

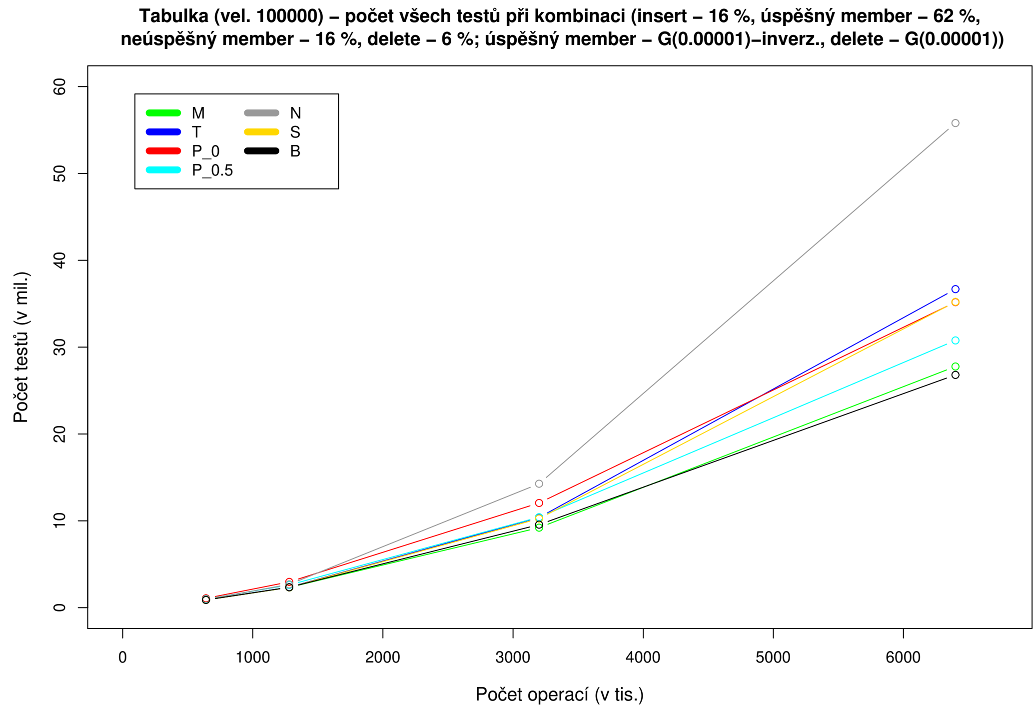
U této možnosti má nejmenší počet testů na vyhledání BVS a na druhém místě je tentokrát TS(0), následovaná strategiemi TS(0.5) a MTF, které mají počet testů na vyhledání v podstatě (navzájem) stejný. V počtu všech testů už je ale TS(0) horší než MTF, a dokonce pro nižší počty operací je na úrovni klasické strategie, která je na počet testů na vyhledání nejhorší. Strategie Sorted je v počtu všech testů pro menší počty operací lepší než TS(0) a TS(0.5), ale pro větší počty operací už horší než obě uvedené strategie. Naměřené časy vcelku odpovídají počtům všech testů, pouze TS(0) a TS(0.5) vycházejí v časech poněkud hůře než v testech.

7.5.5 Zhodnocení

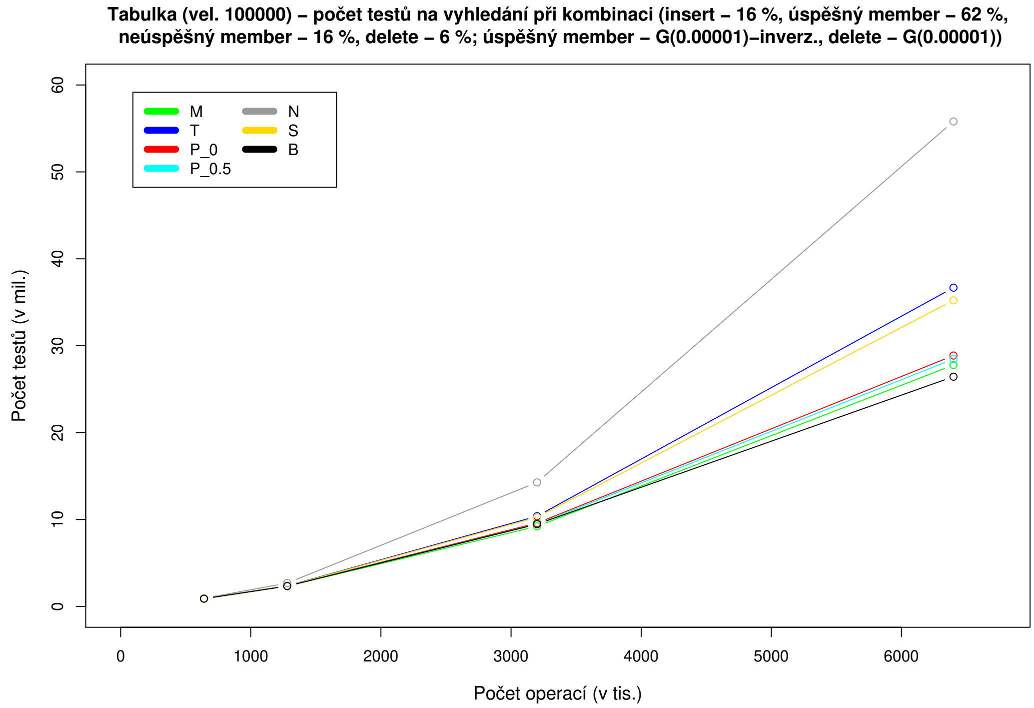
Při testování kombinace měla největší zastoupení operace úspěšný Member a je pravdou, ze výsledky měly jistou podobnost s výsledky experimentů se samotnou operací úspěšný Member s příslušným rozdělením. Přesto jsou zde ale jisté rozdíly. První rozdíl je v tom, že u kombinace má BVS vůči ostatním strategiím lepší



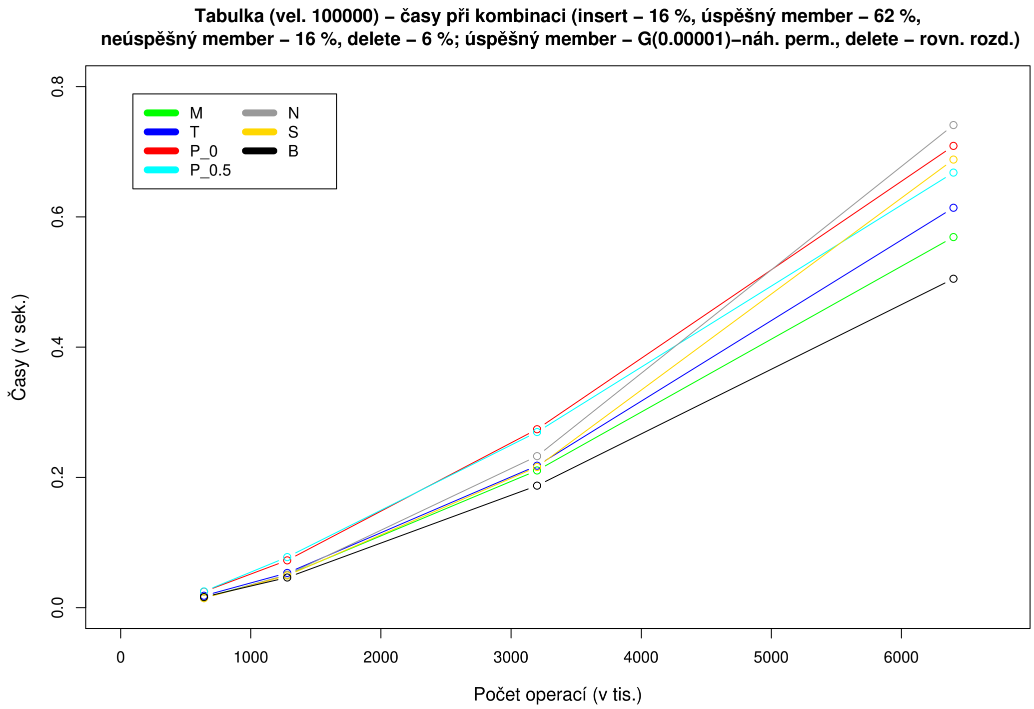
Obrázek 7.30



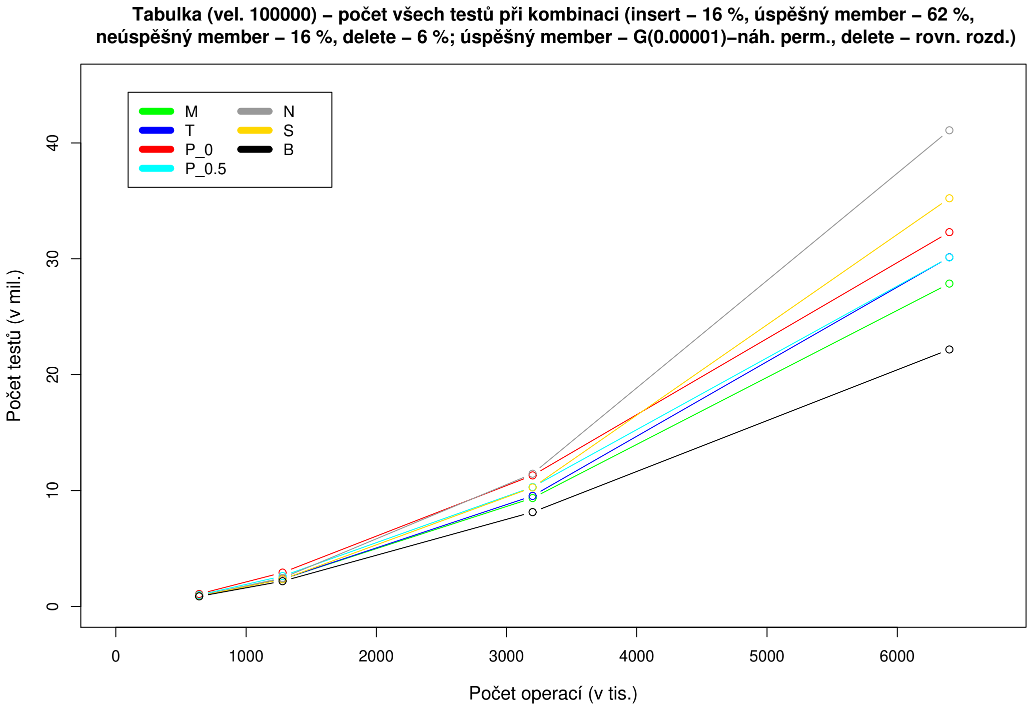
Obrázek 7.31



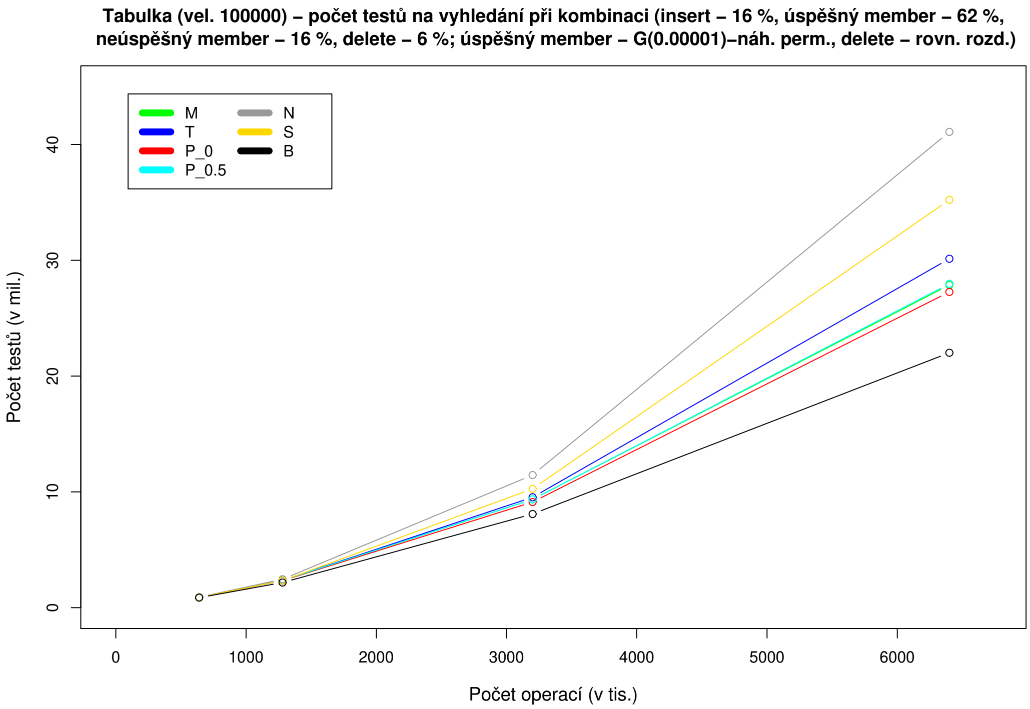
Obrázek 7.32



Obrázek 7.33



Obrázek 7.34



Obrázek 7.35

výsledky než u úspěšného Member (v testech je vždy nejlepší nebo prakticky na úrovni nejlepšího), což je způsobeno převahou nad ostatními strategiemi u Insert a neúspěšného Member. Druhý rozdíl je v tom, ze u samotného úspěšného Member byla při použití rovnoměrného nebo geometrického náhodného rozdělení u klasické strategie a strategie Sorted prakticky stejný počet testů, zatímco u kombinace je jich u Sorted méně. To je způsobeno tím, že Sorted má méně testů u Insert a neúspěšného Member. Ve srovnaní s experimenty pro kombinaci u seznamu jsou vidět asi největší rozdíly u BVS, které nemá na tabulce takovou převahu nad ostatními jako u seznamu, a u TR, která se při nerovnoměrném rozdělení na tabulce (hlavně s nižším faktorem naplnění) chová odlišně od klasické strategie.

Celkově je u kombinace z hlediska počtu všech testů (a tedy i času) nejlepší strategií BVS, druhá nejlepší je asi MTF (bráno celkově, ne všude je druhá nejlepší po BVS).

Kapitola 8

Další výsledky

V této kapitole prezentujeme ještě některé další výsledky, ke kterým se dospělo v průběhu práce. Mohly by se stát i námětem pro nějakou budoucí práci, kde by mohly být podrobněji prozkoumány.

8.1 Vzorce pro BVS u tabulek

Pro náhodné binární vyhledávací stromy existují odhady, které jsme uvedli v kapitole 3.3.2 a ověřili u experimentů se seznamy. Výsledky se velice dobře shodovaly s teorií až snad na očekávanou výšku stromu, kde byl určitý rozdíl.

Zajímavé by bylo zkusit, jestli by tyto odhady nešly nějak využít i u hašovacích tabulek, kde tedy nemáme jeden strom, ale mnoho stromů (pro každý slot hašovací tabulky jeden). Namísto přesného počtu prvků bychom mohli použít očekávaný počet prvků ve stromě, který je roven faktoru naplnění tabulky.

Dá se předpokládat, že kdyby měly všechny stromy v tabulce stejný počet prvků, tak odhady půjdou bez problémů „rozšířit“ i na případ s tabulkou. Toto ale samozřejmě neplatí (i když se berou vkládané klíče z rovnoměrného rozdělení).

U všech následujících ověření navržených odhadů jsme použili pro testování tabulku velikosti 100 tisíc se 2 miliony prvky, u úspěšného vyhledání jsme provedli 5 milionů operací a u neúspěšného 2 miliony operací. Budeme používat označení n pro počet prvků, m pro velikost tabulky a α pro faktor naplnění. Jak už bylo řečeno, vkládané klíče jsou generované z rovnoměrného rozdělení a také vyhledávání se dělá podle rovnoměrného rozdělení. Navržené odhady pro stromy v tabulce dostaneme z původních odhadů pro samotný strom nahrazením α za n a u počtu testů na naplnění tabulky navíc přenásobením velikostí tabulky m.

Očekávaná výška

Navrhovaný odhad: $4.311 \ln(\alpha) - 1.953 \ln\ln(\alpha)$
Výsledek dle odhadu: 10.77
Výsledek experimentu: 6.67

Zde se tedy výsledek experimentu dost odlišuje od navrženého odhadu, ale je třeba poznamenat, že u očekávané výšky byl i u samotného stromu docela velký rozdíl mezi odhadem a výsledkem experimentu.

Počet porovnání na naplnění tabulky (vložení n prvků)

Navrhovaný odhad: $m\cdot(2(\alpha+1)(H_{\alpha+1}-1)-2\alpha)$ (H_n značí n-té harmonické číslo a dá se aproximovat jako $\ln(n)+0.57721$)
Počet porovnání dle odhadu: 7011276
Počet porovnání dle experimentu: 7205935
Poznámka. Počet testů může být maximálně o m větší, to v případě, že v tabulce není žádný prázdný řetězec.

Očekávaný počet porovnání při úspěšném vyhledání

Navrhovaný odhad: $2H_\alpha - 3 - 2H_\alpha/\alpha$
Výsledek dle odhadu: 3.79
Výsledek experimentu: 4.6
Poznámka. Počet testů je v případě úspěšného vyhledání stejný jako počet porovnání.

Očekávaný počet porovnání při neúspěšném vyhledání

Navrhovaný odhad: $2H_{\alpha+1} - 2$
Výsledek dle odhadu: 5.243
Výsledek experimentu: 5.245

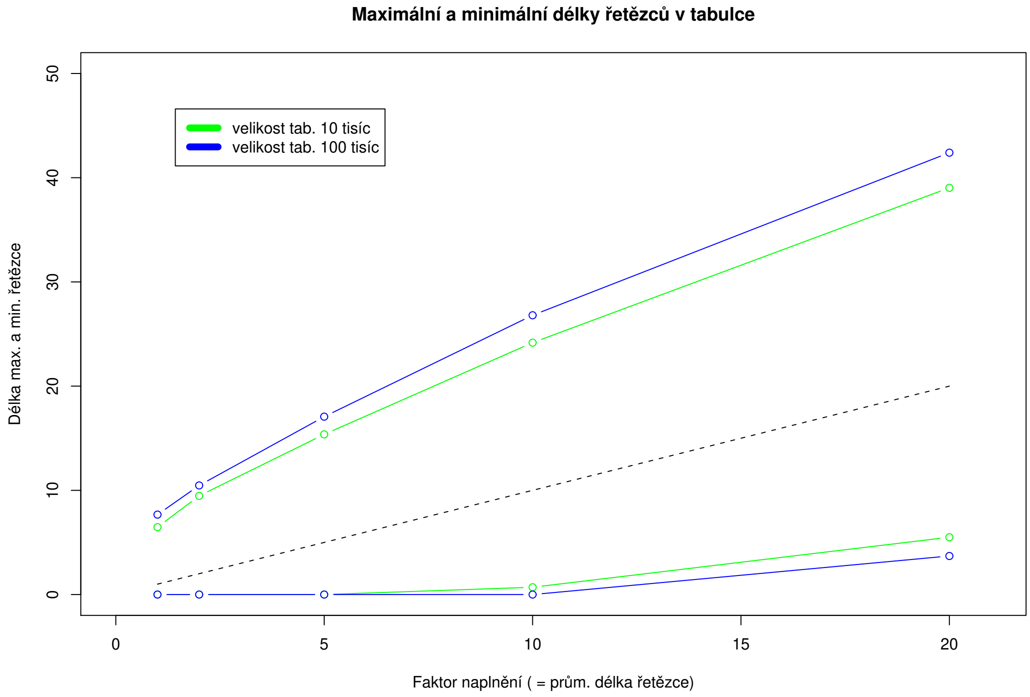
Zde došlo opravdu k velmi dobré shodě. Abychom vyloučili možnost, že se jedná o náhodu, tak jsme ještě provedli srovnání pro tabulky s počtem prvků 500 tisíc a 1 milion. U tabulky s 500 tisíci prvky je odhad 2.74 a výsledek experimentu 2.77 a u tabulky s 1 milionem prvků je odhad 3.95 a výsledek experimentu 3.96. Je tedy vidět, že odhad platí trochu lépe pro vyšší faktory naplnění. Celkově ale lze říci, že navržený odhad (alespoň tedy pro $\alpha \geq 5$) velice dobře odpovídá realitě.
Poznámka. Počet testů je v případě neúspěšného vyhledání obecně větší než počet porovnání, stejný je jen, pokud jsou v tabulce všechny řetězce neprázdné.

Zhodnocení

Před porovnáním navržených odhadů s výsledky experimentů jsme odhadovali, že nerovnoměrnosti v délkách řetězců v kombinaci s tím, že v odhadech jsou logaritmy způsobí, že tyto odhady nebudou správné. Ovšem obdržené výsledky v případě neúspěšného vyhledání se s odhady shodují velice dobře a v případě počtu testů na naplnění tabulky také docela dobře.

8.2 Maximální a minimální délky řetězců

V předchozí podkapitole jsme uvedli, že v případě, že by všechny řetězce v tabulce byly stejně dlouhé, tak by navržené odhady platily pro tabulku stejně dobře jako pro samotný strom. V praxi jsou ale i při použití rovnoměrného rozdělení



Obrázek 8.1

mezi délkami řetězců nezanedbatelné rozdíly, což dokumentujeme v této kapitole. Budeme sledovat maximální a minimální délku řetězců v závislosti na faktoru naplnění.

Maximální délka řetězce je jistě zajímavým údajem, protože pro danou tabulku udává náročnost operace v nejhorším případě. Existuje teoretický odhad pro očekávanou délku maximálního řetězce, který udává horní odhad očekávané maximální délky řetězce při hašování se separovanými řetězci jako $O(\frac{\log n}{\log \log n})$, ale ten se týká pouze případu, kdy je faktor naplnění nejvýše 1. Odhad pro případy s vyšším faktorem naplnění se nám najít nepodařilo.

V následujícím grafu (obr. 8.1) jsou uvedeny maximální a minimální délky řetězců pro tabulky velikosti 10 a 100 tisíc. Navíc je v grafu vyznačena čárkovanou čarou průměrná délka řetězce. V grafu lze vidět celkem očekávaný jev, že při stejném faktoru naplnění je u větší tabulky maximální délka větší a minimální délka menší než u menší tabulky. Z toho lze vyvodit závěr, že očekávaná délka maximálního řetězce nemůže být závislá pouze na faktoru naplnění. Také si můžeme všimnout, že maximální délka řetězce roste pomaleji než lineárně. Minimální délka řetězce je vždy do určitého momentu na nule, což značí, že v tabulce existuje prázdný řetězec. V souvislosti s minulou kapitolou ještě zmiňme, že pokud je minimální délka řetězce větší než nula, tak u neúspěšného vyhledání se počet porovnání shoduje s počtem testů.

Kapitola 9

Závěr

Cílem této práce bylo navrhnout nějaké jiné možné reprezentace řetězců v hašovacích tabulkách než jsou klasické nesetříděné seznamy s neměnným pořadím prvků, implementovat je a porovnat rychlost základních (slovníkových) operací. Hašovacích metod existuje velké množství (v kapitole 2 jsme uvedli deset, a to byla jen část všech existujících). Samozřejmě ne všechny jsou založeny na principu řetězců (příkladem takové, která není, je třeba dvojité hašovaní), ale je jich velká část. Aby bylo časově zvládnutelné provedení dostatečného množství experimentů u více metod, byla nakonec v této práci zkoumána pouze metoda separovaných řetězců.

Základní metodou, jak reprezentovat řetězec, je nesetříděný spojový seznam s neměnným pořadím prvků. Tuto možnost jsme nazvali klasickou strategií. Jako alternativy byly navrženy setříděný seznam (strategie Sorted), náhodný binární vyhledávací strom (strategie BVS) a potom různé typy samoupravujících seznamů. Základní použité samoupravující strategie byly tři (MTF, TR, TS(p)), ale TS(p) navíc díky proměnnému parametru nabízí celou škálu možných algoritmů. Z nich byly pro testy vybrány TS(0) a TS(0.5).

U samoopravujících strategií se zatím snad všechny teoretické výpočty (a i většina experimentů) týkaly pouze samotného seznamu. To nás vedlo k tomu zkoumat i samotné seznamy, porovnat výsledky s teorií a následně také s výsledky obdobných experimentů pro hašovací tabulky. Testovali jsme operace Insert, úspěšný Member, neúspěšný Member, Delete a kombinaci těchto operací. Testovali jsme rovnoměrné rozdělení a u operací úspěšný Member a Delete navíc ještě nerovnoměrné rozdělení, konkrétně geometrické.

Při provádění experimentů jsme měřili počet testů na vyhledání, počet všech testů a čas. Sledovat počet testů na vyhledání je dobré kvůli tomu, že tento ukazatel se vyskytuje v teoretických výpočtech, a proto nám umožňuje porovnání výsledků experimentů s teorií. Naměřeným časům ale lépe odpovídají počty všech testů. To se dá označit i za jeden z přínosů této práce, že se podařilo najít ukazatel složitosti, který lépe odpovídá skutečným časům. Výsledky jsme prezentovali ve formě grafů. V práci nejsou zařazeny grafy ke všem provedeným experimentům, ale všechny jsou k dispozici na přiloženém DVD.

Největší pozornost byla věnována operaci úspěšného vyhledání. Výsledky pro tuto operaci proto probereme trochu podrobněji. Pro strategii Sorted jsou výsledky stejné pro všechna rozdělení, a pokud budeme z těch nerovnoměrných brát v úvahu jen to nezávislé na pořadí vkládání (nazývané náhodné), platí to samé i pro klasickou strategii a BVS. Co se týče samoupravujících strategií, tak ty se na nerovnoměrných rozděleních chovají lépe než na rovnoměrném, když ne hned, tak od určitého počtu operací. Odkdy tomu tak je, záleží na různých faktorech – daném rozdělení, typu samoupravující strategie, vztahu rozdělení k pořadí prvků v seznamu (resp. seznamech tabulky), naplnění seznamu (resp. tabulky). Pokud jsou řetězce delší, tak konvergence (proces přibližování průměrné ceny k asymptotické očekávané ceně) trvá déle. Nejrychleji konvergující strategií je MTF, o něco pomalejší je TS(p) a výrazně nejpomaleji konverguje TR, především na dlouhých řetězcích (viz seznamy – porovnání s OSO – 6.2.5). Na druhou stranu TR má nižší asymptotickou cenu než MTF, takže od určitého počtu operací se už chová lépe než MTF. V počtech testů na vyhledání se jako nejlepší samoupravující strategie jeví TS(0) kombinující rychlou konvergenci s nízkou asymptotickou cenou. Problém u TS(0) je v tom, že je tam velice náročná úprava seznamu, a tak výsledný čas už zdaleka nebývá nejlepší. Z hlediska počtu všech testů a času je nejlepší samoupravující strategií MTF, při hodně velkém počtu operací je pak MTF překonána strategií TR.

Získané závěry by se daly shrnout následujícím způsobem.

- Experimentální výsledky se až na výjimky shodovaly s teoretickými odhady.
- Strategie BVS byla pro seznamy jednoznačně nejlepší a pro tabulku byla překonána pouze při úspěšném vyhledání podle nerovnoměrného rozdělení inverzního nebo náhodného, a to některými samoupravujícími strategiemi.
- Samoupravující strategie – jejich výhoda se projevuje na nerovnoměrných rozděleních při úspěšném vyhledání jak na seznamech, tak na tabulkách. Ze samoupravujících strategií se více osvědčily deterministické, většinou byla nejrychlejší MTF, při opravdu velkém počtu úspěšných vyhledání byla nejlepší TR.
- Neuspořádaný lineární seznam se osvědčil pouze v případech, kdy se jednalo o kratší řetězce nebo rovnoměrné rozdělení, nebo když se nejčastěji vyhledávaly prvky, které byly vloženy nejdříve.
- Uspořádaný seznam se obvykle držel někde uprostřed pořadí zkoumaných strategií, jeho výhody se projevily při vkládání a neúspěšném vyhledání.

9.1 Náměty pro budoucí práci

V této práci jsme se zabývali rovnoměrným a geometrickým rozdělením. Jistě by stálo za to prozkoumat ještě jiné typy nerovnoměrných rozdělení. Také by bylo přínosné vyzkoušet algoritmy na reálných datech, a to nejen číselných, ale třeba na řetězcích. Vybrali jsme pro experimenty jen několik samoupravujících strategií, proto další možností zkoumání by bylo vyzkoušet ještě jiné samoupravující strategie (např. Combination, Frequency Count, ...). Zajímavé by mohlo být také provést experimenty s jinými typy stromů než náhodný binární vyhledávací strom, jako např. AVL-stromy, červeno-černé stromy, atd., zvláště pak vyzkoušet u stromů samoupravování a použít např. Splay-stromy. Dalším námětem na budoucí práci by mohlo být prozkoumání navržených alternativních reprezentací u jiných hašovacích metod než hašování se separovanými řetězci.

Literatura

[1] S. Albers. Improved randomized on-line algorithms for the list update problem. V Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, str. 412–419, 1995.

[2] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. In Proc. of the 23rd International Colloquium on Automata, Languages and Programming, Springer Lecture Notes in Computer Science, Volume 1099, pages 514–525, 1996

[3] S. Albers and J. Westbrook. Self Organizing Data Structures. V A. Fiat, G. J. Woeginger, "Online Algorithms: The State of the Art", Lecture Notes in Comput. Sci., 1442, Springer, Berlin, 13-51, 1998.

[4] S. Albers, B. von Stengel, and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. Information Processing Letters, 56:135–139, 1995.

[5] C. Ambühl, B. Gärtner, and B. von Stengel (2000), Optimal projective list update algorithms. Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP 2000), Lecture Notes in Computer Science 1853, Springer-Verlag, Berlin, 305–316.

[6] R. Bachrach, R. El-Yaniv. Online list accessing algorithms and their applications: Recent empirical evidence. Symposium on Discrete Algorithms. Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, Pages: 53–62, 1997

[7] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. Algorithmica 11, 2-14. Preliminary version in Proc. 22nd STOC (1990), 379–386, 1994.

[8] J. L. Bentley and C. C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. Communication of the ACM, 28:404–411, 1985.

[9] J. R. Bitner. Heuristics that dynamically organize data structures. SIAM J. Comput. 8 (1979), 82–110.

[10] F. R. K. Chung, D. J. Hajela, and P. D. Seymour. Self-organizing sequential search and hilbert's inequality. In Proc. 17th Annual Symposium on the Theory of Computing, 217–223, 1985.

[11] G. H. Gonnet, J. I. Munro, and H. Suwanda. Towards self-organizing linear search. In Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 169–174, 1979.

[12] S. Irani. Two results on the list update problem, Information Processing Letters, v.38 n.6, p.301-306, June 28, 1991.

[13] R. Karp and P. Raghavan. From a personal communication cited in [17]

[14] J. McCabe. On serial files with relocatable records. Operations Research, 12:609–618, 1965.

[15] *The R Project for statistical computing*, http://www.r-project.org/

[16] B. Reed. The Height of a Random Binary Search Tree. Journal of the ACM, Vol. 50, No. 3, May 2003, p. 306–332.

[17] N. Reingold, J. Westbrook, and D. D. Sleator. Randomized competitive algorithms for the list update problem. Algorithmica, 11:15–32, 1994.

[18] R. Rivest. On self-organizing sequential search heuristics. Communication of the ACM, 19:63–67, 1976.

[19] R. Sedgewick, P.Flajolet. An introduction to the analysis of algorithms. Addison-Wesley Publishing Company, 1996.

[20] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. Communication of the ACM, 28:202–208, 1985.

[21] B. Teia. A lower bound for randomized list update algorithms. Information Processing Letters, 47:5–9, 1993.

[22] J. F. Vitter, W.-Ch. Chen. Design and analysis of coalesced hashing. Oxford Univ. Press, 1987.

[23] Wagner R. J.: *Mersenne Twister Random Number Generator*, http://www-personal.engin.umich.edu/ wagnerr/MersenneTwister.html

[24] J. Zobel, S. Heinz, H. E. Williams. In-memory hash tables for accumulating text vocabularies. Information Processing Letters 80 (2001) 271–277.

Příloha A

Obsah DVD

Přiložené DVD obsahuje zdrojové soubory k implementaci jednotlivých strategií a ke generování testovacích dat, ukázková testovací data, výsledky experimentů a samotný text této diplomové práce.

Struktura DVD:

- `src` – adresář se zdrojovými soubory
- `gener` – adresář s nagenerovanými testovacími daty (nejsou zde všechna, jen pro ukázku a možné vyzkoušení)
- `vysledky` – adresář s výsledky experimentů
- `obsah.txt` – soubor s popisem obsahu DVD
- `diplomka.ps` – text diplomové práce

Příloha B

Spouštění experimentů

Zdrojové soubory je možné jednoduše přeložit pomocí volaní (předpokládá se aktuální adresář `src`)

```
make all
```

Pro provádění experimentů slouží skripty testuj_hrom_*název_operace*.sh, které se nacházejí v adresáři `src`. Při spuštění skriptu bez parametrů se vypíšou požadované parametry daného skriptu. Zde je jejich souhrnný popis:

- filesCount – počet sad dat, pro které se má experiment udělat
- startIndex – číslo sady dat, kterou začít
- (init)keysCount – počet vložených prvků před samotným experimentem
- tableSize – velikost hašovací tabulky (pro experimenty se seznamy se použije velikost tabulky 1)
- oper*NázevOperace*Count – počet operací
- distribType*NázevOperace* – typ rozdělení (R – rovnoměrné, G – geometrické)
- distribCoef*NázevOperace* – koeficient geometrického rozdělení (bere se v úvahu, jen pokud je zadáno geometrické rozdělení)
- geomDistribVersion*NázevOperace* – varianta geometrického rozdělení (N – klasická, Y – inverzní, R – náhodná)
- strategy – strategie (N – klasická, M – MTF, T – TR, P – TS(p), S – Sorted, B – BVS)
- paramP – parametr p pro TS(p) (při zadání jiných strategií ignorován)
- dir – adresář, odkud se mají načítat testovací data
- doOutput – jestli se mají výsledky zapsat do souboru (Y – ano, N – ne)

Uvedeme příklad možného spuštění experimentu. Vybereme si operaci úspěšný Member. Pro úspěšný Member jsou předepsány parametry filesCount, keysCount, tableSize, operMemberCount, distribType, distribCoef, geomDistribVersion, strategy, paramP, dir, doOutput, startIndex (v tomto pořadí). Následující volání bude fungovat, pokud aktuálním adresářem bude `src`.

`./testuj_hrom_member_uspes.sh` 1 500000 100000 500000 `G` 0.00001 `R P` 0.7 `../gener N` 1

Tedy experiment se provede pro první sadu dat, testovací data se berou ze sousedního adresáře `gener`, tabulka má velikost 100 tisíc a obsahuje 500 tisíc prvků, provede se 500 tisíc úspěšných vyhledání dle geometrického rozdělení náhodného s koeficientem 0.00001, při vkládání i vyhledání se použije strategie TS(0.7) a výsledky se nebudou zapisovat do souboru, pouze se zobrazí na obrazovce.