Kaštalogizér
Doprovodný dokument

Autor: Jan Hartman
Garant: RNDr. Tomáš Holan, Ph.D.

Studium: bakalářské, 1. ročník, zimní semestr 2022/2023
Předmět: Programování 1

4. února 2023

Obsah

1 Úvodní část **2**
1.1 Anotace . . . 2
1.2 Přesné zadání . . . 2

2 Algoritmická část **2**
2.1 Zvolený algoritmus . . . 2
2.2 Vážení kaštanů v souvislosti s teorií her . . . 2
2.3 Optimalizace algoritmu . . . 3
2.4 Odstranění izomorfních stavů . . . 3

3 Implementace prohledávání stromu hry **3**
3.1 Tah hráče a tah váhy . . . 3
3.2 Implementace tahu hráče . . . 4
3.3 Implementace tahu váhy . . . 4

4 Reprezentace potřebných dat **4**
4.1 Vztahy mezi váhami kaštanů . . . 4
4.2 Zaručení konečnosti . . . 5
4.3 Započítávání tranzitivity . . . 5
4.4 Reprezentace získaných vztahů . . . 5

5 Reprezentace optimalizačních dat **5**
5.1 Stejné kombinace vážení v jiném pořadí . . . 5
5.2 Uchovávání a reprezentace identických stavů hry . . . 6
5.3 Generování kódů označujících identické stavy hry . . . 6

6 Uživatelská část **6**
6.1 Základní vstup . . . 6
6.2 Předvolba vlastních vážení . . . 6
6.3 Výstup a ukončení programu . . . 6
6.4 Testovací vstupy . . . 6

7 Závěr **7**
7.1 Zhodnocení průběhu práce . . . 7
7.2 Druhý závěrečný povzdech . . . 7

1 Úvodní část

1.1 Anotace

Tento program řeší problém hledání určitého kaštanu s konkrétní váhou mezi nějakým počtem kaštanů, kde každý z nich má různou váhu. Tento kaštan smíme hledat pouze za pomoci dvouramenné váhy. Program nám vypočítá nejmenší možný počet vážení na této váze potřebný k tomu, abychom daný kaštan nalezli.

1.2 Přesné zadání

Každá úloha se programu zadává jako posloupnost tří čísel. První číslo udává celkový počet kaštanů, který vážíme. Druhé číslo udává minimální počet kaštanů, které jsou lehčí než námi hledaný kaštan. Třetí číslo udává minimální počet kaštanů, které jsou těžší než daný kaštan. Kaštan, který hledáme pak musí splňovat tato dvě kritéria.

2 Algoritmická část

2.1 Zvolený algoritmus

K řešení problému jsem využil minimaxový algoritmus, který se obecně využívá k prohledávání stromu hry (V našem případě tedy jde nejspíše o statný opadavý strom - jírovec maďal.). Jedná se o hru se dvěma hráči, kde prvním hráčem je hráč MIN, který vybírá dvojici kaštanů, kterou bude vážit. Druhým hráčem, kterému budeme říkat MAX je váha, která vybírá, který z kaštanů je lehčí a který těžší. Cílem hráče MIN je minimalizovat počet vážení k nalezení daného kaštanu. Cílem hráče MAX je tento počet naopak maximalizovat. Předtím, než blíže popíši svou implementaci minimaxového algoritmu, tak by bylo dobré zdůvodnit, proč je problém vážení kaštanů dobře definovanou hrou z teorie her a dává tedy smysl na jeho řešení užít minimaxový algoritmus.

2.2 Vážení kaštanů v souvislosti s teorií her

V následujících odstavcích zdůvodním tři hlavní vlastnosti tohoto problému z pohledu teorie her:

1. je konečnou hrou
2. je s nulovým součtem
3. je s úplnou informací

1. Problém hledání konkrétního kaštanu je konečnou hrou z pohledu teorie her, jelikož kaštanů je konečný počet, a tudíž i možných dvojic na výběr je konečný počet. Tím pádem i počet možných tahů hráče MIN v každém kole je konečný. Hráč MAX má v každém kole jen dva možné tahy. Celá partie je konečná, jelikož hráč MIN nemá povoleno vybrat na zvážení dva kaštany, o jejichž váze už má informaci z předchozích vážení nebo logickou dedukcí z tranzitivity, a tudíž mu dříve nebo později dojdou možné tahy. Hra je u konce, pokud je hledaný kaštan nalezen, což také vždy musí nutně nastat, jelikož pokud by hráč MIN postupně vybral na zvážení všechny dvojice kaštanů, tak má úplnou informaci o jejich uspořádání.

2. Hra je také s nulovým součtem, jelikož čím menší je výsledný počet tahů hráče MIN, tím horší je to výsledek pro hráče MAX.

3. V neposlední řadě se jedná o hru s úplnou informací, jelikož oba hráči mají k dispozici informaci o tom, jaké byly hrány tahy druhým hráčem a také vždy ví, jaké přesně tahy bude mít druhý hráč k dispozici.

Jedná se tedy o dobře definovanou hru dle teorie her a z toho důvodu na ni lze použít minimaxový algoritmus.

2.3 Optimalizace algoritmu

Klasický minimaxový algoritmus jsem v první řadě obohatil o známou techniku alfa-beta prořezávání, což jsem měl v plánu od počátku a nápad mi byl zřejmý.

Následně jsem také odstranil zbytečné procházení případů, kdy si hráč MIN volí dva zatím neznámé kaštany (tzn. nemáme o nich zatím žádnou informaci.) v tomto případě stačí tento výběr v tomto tahu udělat jenom jeden a zbylé takové výběry už rekurzivně neprohledávat. Obecnějším případem tohoto je také stav, ve kterém si hráč vybere nám známý kaštan a váží ho s nějakým neznámým. Pokud toto nastane, tak nezáleží, jaký z neznámých kaštanů s ním budeme vážit, jelikož z hlediska charakteru kaštanů, co se informací o nich známých týče jsou zaměnitelné a povedou ke stejným výsledným stavům. Těmto stavům, kdy vážíme buď dva neznámé nebo jeden známý s nějakým neznámým a nezáleží na tom, jaký z neznámých kaštanů konkrétně vybereme říkám izomorfní stavy a implementaci jejich odstranění popíši v sekci 2.4.

Dalším zkrácením chodu programu je zapamatovávání si průběžných výsledků (bodových ohodnocení) jednotlivých větví programu a následné užití těchto zapamatovaných hodnot, pokud narazíme na případ, kdy byla provedena stejná vážení akorát v jiném pořadí. V tom případě už výsledek známe z případu, kdy jsme ho pro nějaké jiné pořadí vážených kaštanů už spočítali (viz. 5.1).

Poslední důležitou optimalizací bylo odstranění větví, ve kterých mohla váha rozhodnout bez újmy na obecnosti pouze jedním způsobem a druhý bylo možno ignorovat. Tento případ nastává, pokud oba kaštany které se chystáme zvážit jsou neznámé. Implementaci snad ani není nutné blíže popisovat, jedná se pouze o využití datových struktur *mnozinaTezsich* a *mnozinaLehcich* k ověření, zda oba kaštany z vybrané dvojice mají nulový vstupní a výstupní stupeň.

2.4 Odstranění izomorfních stavů

Izomorfní stavy detekuji v těle funkce *tahHrace* za pomocí datových struktur *mnozinaTezsich* a *mnozinaLehcich*, ze kterých lze jednoduše v konstantním čase vyčíst vstupní a výstupní stupně našich kaštanů, kde uvažujeme orientovaný graf, ve kterém šipky vedou od lehčího směrem k těžšímu kaštanu. Pokud oba kaštany mají jak vstupní, tak i výstupní stupeň nula a zároveň platí, že jsme už jinou dvojici se stejnou vlastností vybrali, tak můžeme tuto větev odseknout.

Situace je o něco složitější, pokud se chystám zvážit určitý známý kaštan s nějakým neznámým. V tomto případě jsou totiž neznámé kaštany zaměnitelné, zatímco ty známé nikoliv (mohou mít různé stupně a tedy různý charakter). Je tedy nutné si pamatovat, které z nám již známých kaštanů už jsme s nějakým neznámým kaštanem vážili. Tuto informaci jsem se rozhodl reprezentovat datovou strukturou *vybraneZname* typu `set`. To z toho důvodu, že zkontrolovat zda-li množina obsahuje nějaký prvek či nikoliv lze v asymptoticky konstantním čase. Na druhou stranu celkový počet kaštanů není veliký (Takže lineární čas by nám ani tolik nevadil.), ale jelikož jsem neviděl důvod, proč bych chtěl jednotlivé kaštany v této datové struktuře indexovat, tak mi přišel `set` jako elegantnější volba.

3 Implementace prohledávání stromu hry

3.1 Tah hráče a tah váhy

Prohledávání stromu hry je zajištěno postupným střídavým rekurzivním voláním funkcí *tahHrace* a *tahVahy* přičemž hra začíná tahem hráče MIN, tedy první z funkcí. Oběma funkcím předáváme tato data:

1. *mnozinaTezsich* a *mnozinaLehcich* (viz. 4.3)
2. *zbyleKombinace* (viz. 4.4)
3. *pocetVazeni* (viz. 3.2)
4. *provedenaVazeni* (viz. 3.2)
5. *binarni* (viz. 5.3)
6. *alfa* a *beta* (viz. 2.3)

Pojďme si objasnit, jak tyto dvě funkce s těmito daty pracují a jak si je předávají.

3.2 Implementace tahu hráče

Tato funkce reprezentuje tah hráče MIN a její hlavní úlohou je vracet na vyšší hladinu nejpříznivější volbu tahu pro tohoto hráče. Konkrétně vrací trojici proměnných (stejně tak jako funkce *tahVahy*), kde první položkou je počet provedených tahů, druhou je index nalezeného kaštanu a třetí je posloupnost provedených vážení. Který tah tato funkce vyhodnotí jako nejpříznivější se rozhoduje na základě první položky vrácené trojice. Počet tahů v nejpříznivější situaci zaznamenávám v proměnné *alfa*, jejíž hodnotu pak dále předávám funkci *tahVahy*, abych ji mohl využít k implementaci alfa-beta prořezávání.

Důležité je také říci, že funkce *tahHrace* přijímá výše zmiňovanou trojici dat vždy od funkce *tahVahy* a nikdy ji sama negeneruje. To hlavně z toho důvodu, že konečný stav, kdy nalezneme hledaný kaštan nastane až poté, co váha vybere, který kaštan vyhodnotí jako těžší, a který jako lehčí. Samotný výběr dvojice kaštanů hráčem MIN ještě nevede k získání nové informace o váhách kaštanů a proto jsem se rozhodl v této funkci ani nijak neupravovat předávané datové struktury *mnozinaTezsich*, *mnozinaLehcich* a *zbyleKombinace*. Funkce *tahHrace* tyto datové struktury tedy pouze předá dále nedotčené a nevytváří tím pádem žádné nové kopie.

Informaci o tom, kterou dvojici kaštanů si hráč MIN v aktuální větvi stromu hry vybral na zvážení se předává pomocí proměnné *provedenaVazeni*, která je typu `tuple` a skládá se z uspořádaných dvojic, které odpovídají doposud zváženým dvojicím kaštanů (dvojice jsou také typu `tuple`). Tuto posloupnost pak funkce prodlouží o právě vybranou dvojici kaštanů a předá ji funkci *tahVahy*. Dvojice, ze kterých si hráč MIN v těle této funkce může vybírat jsou uloženy v datové struktuře *zbyleKombinace*, kterou blíže popisuji v sekci 4.4. Tuto strukturu procházím pomocí prostého `for` cyklu přes všechny její prvky, přičemž některé průchody tímto cyklem přeskakuji způsobem, který popisuji výše (viz. 2.4).

3.3 Implementace tahu váhy

Druhou funkcí účastnící se rekurze je funkce *tahVahy*, která má za úkol vrátit hráči MIN na vyšší hladinu stromu hry co možná nejnepříznivější výsledek vážení. Tato funkce se od tahu hráče liší především tím, že právě v této funkci je prováděn výpočet tranzitivity voláním funkce *zvazAktualizujMnoziny AVratHledanyKastan*. Tato funkce vrací hodnotu -1, pokud upravení vztahů (viz. 4.3) nevedlo k odhalení hledaného kaštanu, nebo naopak index nalezeného kaštanu. Tuto hodnotu pak můžeme využít při kontrole, zda-li nastal koncový stav (báze této rekurze).

Kvůli tomu, že funkce *zvazAktualizujMnoziny AVratHledanyKastan* upravuje datové struktury *mnozinaTezsich*, *mnozinaLehcich* a *zbyleKombinace*, tak je nutné před jejich zavoláním vytvořit nové kopie těchto struktur, které následně budeme předávat funkci *tahHrace* (pokud se již nenacházíme v koncovém stavu).

V těle této funkce také implementuji dvě optimalizační techniky, které zmiňuji v ostatních sekcích. Za prvé odsekávání větví, ve kterých nezáleží na rozhodnutí váhy (viz. 2.3) a za druhé předpočítávání si již známých výsledků pro stejné neuspořádané n-tice dvojic kaštanů akorát v jiném pořadí (viz. 5.1).

4 Reprezentace potřebných dat

4.1 Vztahy mezi váhami kaštanů

Nejprve zmíním, která data bude v programu nutná reprezentovat a následně se každé kategorii dat budu věnovat konkrétně. Hlavní roli hraje datová struktura reprezentující informaci, kterou má hráč MIN o váhách kaštanů. Tato datová struktura funguje jako taková tabulka, ve které si v každém úseku hry můžeme pro jakýkoliv kaštan vyhledat, kolik kaštanů je lehčích a kolik těžších než on. Je to důležitá datová struktura pro zaručení konečnosti naší hry, jelikož hra končí právě stavem, ve kterém v tabulce najdeme kaštan s vyhovujícím počtem lehčích a těžších kaštanů. Tato struktura samotná však ještě konečnost nezaručí. Mohlo by se stát, že hráč by pořád dokola vybíral stejnou dvojici kaštanů na zvážení a nikdy by tedy nezískal nové informace o váhách ostatních kaštanů.

4.2 Zaručení konečnosti

Právě tento problém řeší druhá důležitá datová struktura, která zaznamenává neuspořádané dvojice kaštanů, které už hráč vybral na váhu. Tu bychom si zase mohli představit jako kompletní seznam všech možných dvojic kaštanů, ze kterého si hráč postupně odškrtává, které kaštany už vybral na zvážení. Problém však není tak přímočarý, jelikož předpokládáme, že hráč MIN je racionální agent (Využiji této hezké terminologie z teorie her.) a bude si tedy ve svém seznamu odškrtávat i takové dvojice, o jejichž váhách hráč MIN získal informaci logickou dedukcí, i když kaštany samotné na váze nebyly. Např. pokud je kaštan s indexem 0 těžší než kaštan 1 a zároveň kaštan 1 je těžší než kaštan 2, tak víme, že kaštan 0 je těžší než kaštan 2, i když tyto dva kaštany jsme na váhu vůbec nepokládali. Tomuto vztahu říkáme tranzitivita a přesně tento jev budu myslet dále v textu, pokud budu tranzitivitu zmiňovat.

4.3 Započítávání tranzitivity

Tranzitivitu započítávám během tahu váhy, tedy v těle funkce *tahVahy* hned poté, co váha určí, který z kaštanů vyhodnotí jako lehčí, respektive těžší. Konkrétní funkce, která se o tento proces stará je *zvazAktualizujMnozinyAVratHledanyKastan* a k vyhodnocování tranzitivity využívá informaci o váhách kaštanů, které vyčte z datových struktur *mnozinaTezsich* a *mnozinaLehcich*. Pro lepší orientaci v kaštanech nyní označme kaštan, který v tomto vážení byl vyhodnocen jako těžší „vítěz“ a lehčí jako „poražený“. Nejprve funkce `for` cyklem projede všechny kaštany, které jsou ještě těžší, než je „vítěz“ z právě zvážené dvojice. Těmto kaštanům budu říkat „nadvítězové”. Protože „nadvítězové” musí z tranzitivity být také těžší než „poražený“, tak pomocí funkce *pridejDoMnozinAVratHledany* upravíme vždy množiny lehčích a těžších u „poraženého“ a odpovídajícího „nadvítěze”. Zároveň však mohou existovat kaštany, které jsou lehčí než „poražený“. Těmto kaštanům budeme říkat „podporažení“. Nyní je tedy ještě potřeba pro každého „podvítěze“ upravit vztah mezi jeho váhou a váhami „nadvítězů“ stejným způsobem.

Samotný zápis získaných dat o váhách kaštanů pak provádí funkce *pridejDoMnozinAVratHledany*, která zároveň při této příležitosti rovnou zkontroluje, zda-li úpravou vztahů mezi kaštany nebyl odhalen právě námi hledaný kaštan. Přišlo mi dobré, kontrolovat tuto skutečnost právě na tomto místě v algoritmu, jelikož jinak bychom museli někde dále v těle funkce *tahVahy* otestovat existenci kaštanu s požadovanými kritérii prohledáním obou struktur *mnozinaTezsich* a *mnozinaLehcich*, což by nám zabralo asymptoticky lineární množství času ku celkovému počtu kaštanů.

4.4 Reprezentace získaných vztahů

Nyní popíšu více do hloubky datovou strukturu zmiňovanou v sekci 4.2, která hráči MIN říká, které dvojice kaštanů má smysl vybrat na zvážení, a naopak o kterých už má informaci odhalenou. Té v programu říkám *zbyleKombinace* a jedná se o datový typ `set`. Tento datový typ jsem zvolil z toho důvodu, že odebírat z něj prvky dokážeme s konstantní asymptotickou časovou složitostí a zároveň nám nezáleží na pořadí. Aby hráč měl z čeho odebírat, tak musím na začátku běhu programu do této množiny vložit všechny možné neuspořádané množiny kaštanů. O to se stará funkce *vygenerujKombinace*.

5 Reprezentace optimalizačních dat

5.1 Stejné kombinace vážení v jiném pořadí

Důležitými optimalizačními datovými strukturami mého programu jsou datové struktury starající se o to, aby program zbytečně neprohledával takové větve hry, které odpovídají stejným kombinacím provedených vážení akorát v jiném pořadí, než v aktuálně prohledávané větvi. Jedná se totiž o identický stav hry, co se týče vztahů mezi váhami kaštanů (Ty budou stejné nehledě na to, v jakém pořadí jsme informace o jejich váhách informace získávali.), i když se jedná o jinou větev rekurze, kterou bychom potenciálně museli celou zbytečně znovu prohledat. To je však nežádoucí a z toho důvodu je nutné zavést systém, kterým budu tyto případy pokud možno v co nejrychlejším čase identifikovat. K tomuto účelu slouží v mém programu datové struktury *spocitaneKombinaceVazeni* a *kodovaniDvojic*. Obě tyto struktury jsou typu `dict` (slovník) a postupně si je rozebereme v následujících sekcích.

5.2 Uchovávání a reprezentace identických stavů hry

Na uchování identických stavů jsem využil datový typ `dict`, jelikož ten umožňuje v konstantním čase zkontrolovat, zdali už se daný stav ve slovníku nachází nebo ne a případně ho v konstantním čase do slovníku přidat. Do slovníku však nebudu ukládat přímo posloupnosti provedených vážení, jelikož jejich přímým porovnáním nezjistím, jestli se jedná o identický stav hry. Je tedy nutné najít metodu, díky které každé posloupnosti provedených vážení přiřadím nějaký kód, který bude stejný pro všechny další posloupnosti provedených vážení, které jsou pouze nějakou její permutací. Právě tyto kódy pak budu vkládat do slovníku *spocitaneKombinaceVazeni*. Jak tyto kódy generuji popíši v další sekci.

5.3 Generování kódů označujících identické stavy hry

Hlavní myšlenkou za tímto kódem je fakt, že si posloupnost provedených vážení můžu představit jako číslo zapsané ve dvojkové soustavě. Každá pozice cifry tohoto čísla pak odpovídá nějaké uspořádané dvojici kaštanů přičemž pokud se tato dvojice kaštanů nachází v naší posloupnosti provedených vážení, tak bude na této pozici jednička, jinak nula. Pokud mám tedy dvě posloupnosti provedených vážení, které jsou však různě zpermutované, tak jim bude odpovídat stejné číslo ve dvojkové soustavě, čímž zjistím, že se jedná o identické stavy hry.

Každé uspořádané dvojici kaštanů tedy musím přiřadit nějaké unikátní číslo, které říká, která pozice binárně zapsaného čísla mu odpovídá (Což je de facto to samé jako říci, které mocnině dvojky tato uspořádaná dvojice odpovídá). Toto přiřazení stanovuji v těle funkce *konfigurujKodovaniDvojic* a jeho „mapu“ reprezentuji slovníkem *kodovaniDvojic*. Samotou hodnotu odpovídajícího čísla pro danou větev stromu hry si pak pamatuji v proměnné *binarni*.

6 Uživatelská část

6.1 Základní vstup

Po spuštění programu je nutné na vstupu zadat uspořádanou trojici kladných celých čísel na jednom řádku, kde každé z čísel je odděleno mezerou. Po zadání všech třech čísel je nutné volbu potvrdit stisknutím tlačítka enter.
Následuje pět vstupů, které vyjadřují volbu ano/ne přičemž volbě „ano“ odpovídá číslo 1 a volbě „ne“ odpovídá číslo 0. Každý z těchto vstupů je nutné potvrdit stisknutím klávesy enter.

6.2 Předvolba vlastních vážení

Speciálním případem je vstup, který je od uživatele očekáván, pokud jako pátý z výše zmíněných vstupů zadá 1 (Tedy na poslední otázku odpoví „ano“.). V tom případě je od uživatele očekávána na vstupu posloupnost indexů kaštanů sudé délky. Jednotlivé indexy kaštanů musí být odděleny mezerou a zadávají se všechny na stejný řádek. Počínaje prvním zadaným indexem každá dvojice indexů odpovídá dvojici kaštanů, které jsme proti sobě vážili. První prvek této dvojice označuje kaštan, který váha vyhodnotila jako těžší. Vstup je nutné následně potvrdit stisknutím tlačítka enter.

6.3 Výstup a ukončení programu

Program následně provede svůj výpočet. Je však nutné počítat s tím, že doba běhu výpočtu pro větší počet kaštanů, než je sedm, může být i větší než v řádech hodin.
Po doběhnutí výpočtu program vytiskne výsledek a k němu i testovací statistiky, pokud si je uživatel vyžádal. Následně je možné výpočet provést znovu akorát s jinými přednastavenými váženími, pokud uživatel na vstupu zadá číslo 1. V opačném případě program je program ukončen.

6.4 Testovací vstupy

Zde ještě přidávám pro uživatele bez inspirace sadu několika testovacích vstupů:

- Nejtěžší ze tří - Vstup: 3 2 0, Očekávaný výsledek: 2
- Prostřední z pěti - Vstup: 5 2 2, Očekávaný výsledek: 6
- Nejtěžší z pěti - Vstup: 5 4 0, Očekávaný výsledek: 4

- Druhý nejtěžší ze šesti - Vstup: 6 4 1, Očekávaný výsledek: 7
- Nejtěžší ze sedmi - Vstup: 7 6 0, Očekávaný výsledek: 6

7 Závěr

7.1 Zhodnocení průběhu práce

Bohužel nemám změřený přesný počet hodin, který jsem prací na programu strávil. Delší dobu, než samotné programování mi ale jednoznačně zabralo vymyslet nápad, jak reprezentovat vztahy mezi váhami kaštanů tak, aby se po každém vážení započítala tranzitivita. Vzal jsem si k srdci doporučení, že je dobré si program rozmyslet dříve, než začnu „bušit“ do klávesnice, a dokonce jsem využil program na myšlenkovou mapu, což jsem během práce na domácích úkolech z programování nebo algoritmizace nepotřeboval. (Většinou mi stačil nějaký méně strukturovaný náčrtek, kterých jsem i při tvorbě tohoto programu vyprodukoval nemálo.) Dlouhou dobu mi také trvalo, než jsem se rozhodl, jaké zvolím konkrétní datové struktury. Konkrétním zajímavým případem bylo také rozhodování, jak co nejefektivněji reprezentovat různé posloupnosti vážení akorát v jiném pořadí, tak, abych je pak mohl rychle identifikovat v programu. Zde mi trvalo dlouhou dobu, než mě napadlo, jakým způsobem bych tuto informaci měl v programu uchovávat a doposud nevím, jestli moje kódování je dobrý způsob (Třeba by to šlo nějak mnohem elegantněji.).
Opačným případem však byla finální úprava programu tak, aby byly ošetřeny všechny možné špatné vstupy od uživatele a aby program nekončil chybovým hlášením po zadání špatného vstupu uživatelem. Při práci na tom jsem delší čas věnoval naopak samotnému programování a čeho chci docílit mi bylo jasné hned.

7.2 Druhý závěrečný povzdech

Pobídka k tomu, abych detailněji rozepsal svou programátorskou část dokumentace pro mě byla velmi přínosnou zkušeností a překvapivě mě vedla k celkem razantním změnám v programu, které vedly k velkému zlepšení čitelnosti a kupodivu i jeho rychlosti výpočtu. Zjistil jsem, jak je výhodné mít seznam testovacích dat, pokud člověk na programu dělá malé změny a chce si být jistý, že jimi nic nezkazil. Stačí pak rychle provést všechny testy a je celkem velká šance, že jsme nic nezkazili. Zkrátka mě samotného překvapilo, na kolik nápadů se dá přijít podrobným rozepisováním toho, co každá část programu konkrétně dělá. Dokonce jsem si v pozdních fázích psaní této druhé verze dokumentace musel sám slíbit, že už budu opravdu jen dokumentovat a nezačnu si program znovu měnit pod rukama. Byla to zábava!