Statement of collaboration

John har arbetat mer djupgående med implementationen och kodandet, även om vi skrivit delar av koden tillsammans. Tråkigt nog är koddelarna av projektet bitvis uteslutna ur rapporten, men finns tillgängliga vid intresse. Rapporten är till stor del skriven av bägge två även om Anton mer fokuserat på Inledning, Problembeskrivning och Slutsatser och John mer fokuserat på Bakgrund, Tillvägagångssätt och Resultat.

Analys av tangentbordsutformningen

Ur programmerarens perspektiv

JOHN BRYNTE TURESSON
ANTON ERHOLT

Kandidatexamensrapport vid CSC/TMH
Handledare: Roberto Bresin
Examinatorer: Karl Meinke & Mads Dam

TRITA xxx yyyy-nn

Referat

Utformningen av tangentbordet har sett annorlunda ut genom tiderna, men de senaste drygt hundra åren har den inte ändrat sig mycket. Alternativa tangentbordsutformningar existerar, men har inte fått det genomslag man kunnat tro de skulle få. Är de verkligen så bra som de säger? Denna rapport utreder hur några vanligt förekommande layouter förhåller sig till varandra ur ansträngningssynpunkt, för några väl valda mjukvaruprojekt skrivna i programmeringspråket Java.

Abstract

Analysis of the keyboard layout from a programmers perspective

The design of the computer keyboard has been different throughout its entire lifespan, but in the last hundred years nothing much have changed. Alternative layouts do exist, but they have not been able to fully reach the market in the way one would expect. Are the alternative layouts really as good as they claim to be? This thesis investigates the relations between a few common keyboard layouts and compares them by means of the effort required to use them, based of a few hand-picked software projects written in the programming language Java.

Innehåll

Kapitel 1

Introduktion

Efter några år på KTH har studenterna hunnit med en hel del skrivande, läsande och programmerande. Både många och långa blir de timmar som studenterna spenderar framför skärmen. De allra flesta inser inte hur utsatta och ansträngda deras kroppar blir i och med den enorma tidsmängd de spenderar framför skärmen och tangentbordet.

När man senare kommer ut på arbetsplatser inser man kanske att det ligger i mångas intresse att ordna sin arbetsplats på ett ergonomiskt vis. Men, trots justerbara stolar och bord är det betydligt färre som använder något annat än ett traditionellt QWERTY-tangentbord när de skapar vid datorn.

Eftersom programmerare ser datorn som ett arbetsredskap, ett skapande verktyg, är det aningen förvånande att man inte brytt sig mer om att effektivisera och optimera de inmatningsmetoder som finns att tillgå. Ett av syftena med denna rapport är därför att utreda om det finns något att vinna med en utveckling av tangentbordslayouten för programmeraren.

Skaderisken i och med långvarig användning av tangentbord är en bidragande faktor till varför just tangentbordslayouter är ett intressant ämne. Det finns självfallet andra faktorer, exempelvis kan man även här komma dragandes med det klassiska plånboksargumentet. Man kan tänka sig att man sparar tid (och därmed även pengar) genom att finna en optimal utformning.

Kapitel 2

Problembeskrivning

Det existerar i nuläget flertal olika tangentbordslayouter såsom QWERTY, DVORAK och Programmer's DVORAK. Trots dess ineffektivitet och långa historia som går tillbaka långt innan datorns uppkomst är QWERTY den överlägset mest använda layouten idag. Varför har ingen annan layout än QWERTY lyckats slå igenom så pass globalt, och hur ser programmerare på tangentbordsstandarden; finns det möjlighet till utveckling?

Många är de frågor som väcks till liv när man börjar rota i ämnet. En av de mest intressanta frågorna man kan ställa sig är ändå:

> Finns det ett behov av att utveckla tangentbordsstandarden för programmeraren?

Följdfrågorna som dyker upp är ej heller de få till antalet och tillsammans konstituerar de rapportens frågeställning. Hur kan man jämföra tangentbordslayouter på ett vettigt sätt? Vilka fysikaliska storheter eller egenskaper hos tangentbordslayouterna kan användas vid jämförelsen? Givetvis kan man också fråga sig vad ett byte av en layout får för konsekvenser, eller vad det får för effekter hos tangentbordsanvändaren. Hur mycket "kostar" det användaren att hålla fast vid sin invanda layout?

Det finns exempel på diverse förslitningsskador[5] som kan uppstå vid långvarit användande av tangentbord. Arbetsmiljöverket rekommenderar olika typer av tangentbord för olika arbetssituationer[6], något som rimligtvis skulle kunna vara intressant även för en programmerare. Om man kan visa att en tangentbordslayout leder till mindre ansträngningar och stress för händerna, kan man då därmed säga att den indirekt bidrar till en bättre livskvalitet för programmeraren?

Kapitel 3

Bakgrund

Vid utformandet av arbetsplatser där en person ska kunna styra och interagera med ett system finns det flera aspekter att tänka på som rör valet av inmatningsmetoder. Att ha knappar med fixa eller variabla funktioner och att placera och gruppera knappar efter betydelse, frekvens, funktion och ordningsföljd är exempel på vad som kan tas med i beräkningarna. [1, s. 123-128] Det finns en uppsättning riktlinjer framtagna av Hanes (1975) som sammanfattar processen att ta fram en tangentbordslayout:

1. Determine the characters and numbers required.
2. Arrange the keys according to their frequency of use and user characteristics. The most frequently used keys should be assigned to the stronger fingers. To enhance keying speed, the keys should also be arranged to maximize alternation of key presses between hands.
3. Follow historical precedent.
4. Follow established standards.
5. Group frequently used keys under resting position of the hand where the user can determine their locations by touch.
6. Group related functions together.
7. Group logically and according to sequence of use.
8. Locate according to importance.
9. Code the keys so that the user can easily locate important or frequently used keys and key groups. In addition to key label, keys can be coded by variations of shape, color, surface texture, and spacing.
10. Consider all factors, including the indented applications, costs, and manufacturing requirements. [1, s. 128-129]

Tangentborden vi har idag är resterna av vad som tidigare var skrivmaskiner. De tidigaste skrivmaskinerna inspirerades av pianots klaviatur[12] och hade bokstäverna utplacerade i alfabetisk ordning. När sedan fler kommersiellt slående skrivmaskiner började dyka upp på marknaden, (i synnerhet maskinen som Christopher Sholes tog patent på 1868), hade placeringen av tangenterna utvecklats långt ifrån den ursprungliga alfabetiska ordningen. [14]

QWERTY, efter den övre radens första sex bokstäver, kallades den tangentplacering som syntes på de allra flesta skrivmaskiner vid den moderna skrivmaskinens genombrott på 1870-talet. Dessa skrivmaskiner hade en uppsättning typarmar, en för varje tangent, som hamrades på pappret när en tangent trycktes ned och gjorde ett avtryck av tecknet på pappret. Ett vanligt förekommande problem var att dessa typarmar fastnade i varandra när intilliggande tangenter trycktes ned samtidigt. En lösning på problemet var att placera tangenter som vanligtvis trycks ned i följd så långt ifrån varandra som möjligt och på så vis ge typarmarna tillräckligt med svängrum. Problemet gav därmed upphov till QWERTY-layouten där vanligt förekommande teckenpar, eller bigram, är uppdelade och placerade långt ifrån varandra. [15]

August Dvorak kom år 1932 med en ny tangentbordslayout som skulle vinna över QWERTY-layouten. DVORAK, som blev tangentbordslayoutens namn, var tänkt att förbättra allt som QWERTY gjorde dåligt. Hanes lista ovan är en bra sammanfattning över de riktlinjer som definierar DVORAK-layouten. Till skillnad från QWERTY utvecklades DVORAK i en tid då skrivmaskinerna inte var lika begränsade vilket bidrog till att layouten kunde utvecklas mer efter frekvensanalyser.

Kapitel 4

Tillvägagångssätt

4.1 Programmeringsspråk

I programmeringsspråk används flera tecken som knappt används alls i vanliga texter men som återfinns i programkod som syntaktiska beståndsdelar. Alla typer av parenteser är bland annat en viktig beståndsdel i programkod. Vanliga parenteser används vid funktionsanrop, klammerparenteser används för att skapa listor och hakparenteser används som olikhetstecken. Gemensamt för alla specialtecken är att de är placerade runt bokstavstangenterna på tangentbordet, som en följd av att tangenterna i mitten redan var upptagna. I Java spelar de svåråtkomliga måsvingarna en speciellt stor roll eftersom de omsluter funktions-, konditions- och klassblock. Måsvingarna, tillsammans med hakparenteserna, är placerade i altgr-läge vilket betyder att altgr-tangenten behöver hållas ned med tummen. Utöver detta sitter speciellt den vänstra måsvingen i mitten av sifferraden vilket gör att handen måste göra en mycket ansträngd sträckning mellan tummen och pekfingret med en udda vinkling på handen. Eftersom Java också är ett starkt typat språk förekommer många nyckelord som en del av syntaxen. Detta gör att samma ord förekommer ofta samtidigt som de vanligt förekommer i en följd.

4.2 Betygssättning av en tangentbordslayout

En tangentbordslayout kan inte enbart tas fram med hjälp av frekvensanalys av tecken i en text. Att samla data om vilka tecken som oftast kommer dyker upp i följd ger en bra bild av hur en text är uppbyggd, men för att bygga en tangentbordslayout utifrån samma data behövs vikter för vad som klassas som en bra layout. Under detta avsnitt diskuteras de viktigaste aspekterna att ta med i beräkningarna för att betygsätta en tangentbordslayout.

4.2.1 Carpalx

Carpalx är ett program framtaget av Martin Krzywinski, som kan ta fram optimerade tangentbordslayouter för specifika användningsområden genom att matas med träningsdata. [**?**]arpalx Träningsdatan, som ges i formen av textdokument, delas upp i trigram (triad på engelska) som i konstrast till bigram är en följd av tre tecken.

Trigramen kan modifieras så de överlappar varandra olika mycket. I Carpalx är dock standardinställningen satt till att överlappa med två tecken (alltså ett nytt trigram för varje tecken). Varje trigram betygssätts med ett effort-värde, eller ansträngningvärde, sammanräknat av hur kostsam teckensekvensen är att skriva för händer och fingrar. Hela tangentbordet får därefter ett sammanlagt betyg genom att ta medelvärdet på alla triaders betyg i hela träningsdatan (se ekvation 4.1).

Applikationen kan också med hjälp av dessa uträkningar hitta en optimal layout baserat på den givna träningsdata genom att flytta runt tangenter och minimera ansträngningsvärdet.

$$E = \frac{1}{N} \sum_i e_i \tag{4.1}$$

Modellen som definierar beräkningen av ansträngningsvärdet i Carpalx kan delas upp i tre huvudsakliga delar: base effort (b), penalty (p) och stroke path (s). Base effort definieras som avståndet fingret färdas för att nå tangenten. Avståndet mäts från fingrets vilotangent på hemraden. Värdet beräknas utifrån alla tre tecken i trigramet och sammanvägs enligt ekvation 4.2. På så sätt blir det sammanlagda värdet beroende av de intilliggande tecknen i teckensekvensen.

$$b_i = k_1 b_{i1}(1 + k_2 b_{i2}(1 + k_3 b_{i3})) \tag{4.2}$$

Penalty definierar en kostnad för varje tecken sammanräknat av använd hand, tangentbordsrad samt använt finger enligt ekvation 4.4. Kostnaden för använd hand kan sättas på så sätt att den ”bättre” handen får en lägre kostnad än den ”sämre”, men vanligtvis sätts inget kostnadsvärde alls händerna. De fyra olika raderna (sifferrad ner till raden ovanför mellanslagstangenten) tilldelas en varsin kostnad där hemraden vanligt tilldelas kostnad 0. Den kostnad som sätts för varje finger baseras på styrkan i varje finger där lill- och ringfinger typiskt får en högre kostnad. Kostnaderna för varje tecken sammanvägs därefter på samma sätt som för base effort enligt ekvation 4.3.

$$p_i = k_1 p_{i1}(1 + k_2 p_{i2}(1 + k_3 p_{i3})) \tag{4.3}$$

Tabell 4.1. Penalty-värden för stroke path

value	hand, p_h	row, p_r	finger, p_f
0	both used not alternating eem ope	same ert als	all different, monotonic progression asd pua
1	alternating aja sot	downward progression, with repetition ern kam	some different, key repeat, monotonic progression app err
2	same ase mon	upward progression, with repetition ade nal	rolling bih fad
3		some differe monotonic, max row change jab oar	all different, not monotonic yak nep
4		downward progression eln pax	some different, not monotonic progression kri maj
5		some different, not monotonic, max row change downward >1 hen kib	same, key repeat cee loo
6		upward progression zaw nap	some different no key repeat, monotonic progression abr bde
7		some different, not monotonic, max row change upward >1 abe axe	same, no key repeat tfb dec

$$p_{ij} = w_0 + w_1 P_{hand} + w_2 P_{row} + w_3 P_{finger} \tag{4.4}$$

Stroke path beräknas utifrån handalternering, radanvändning samt fingeranvändning mellan tecknen i teckensekvensen enligt ekvation 4.5. Kostnaden för händer, rader och fingrar bestäms utifrån de kombinationer av rörelser som kommer som en följd av teckensekvensen. Tabellen nedan visar alla dessa rörelsekombinationer och deras motsvarande kostnad.

$$s_i = f_{hand} p_{hand} + f_{row} p_{row} + f_{finger} p_{finger} \tag{4.5}$$

Slutligen beräknas summan av de tre ansträngningsvärdena vilket ger det sammanlagda ansträngningsvärdet för trigramet enligt ekvation 4.6. De vikter som står intill varje parameter i ekvationerna diskuteras nedan under avsnittet Parametrar.

$$e_i = k_b b_i + k_p p_i + k_s s_i \tag{4.6}$$

4.2.2 Brister med Carpalx

Applikationen är i grunden byggd för att optimera tangentbordslayouter med engelskspråkiga texter. Träningsdatan till de tester utförda av Krzywinski är tagen från engelska bokklassiker och innehåller alltså i stort sett endast bokstavs- och siffertecken samt punktuering och kommationstecken. Modellen lägger därför inte någon större vikt på användningen av funktionstangenter eftersom den största andelen tecken är åtkomliga med shift-tangenterna som enda tillägg. Inte heller användningen av mellanslagstangenten finns med i Krzywinskis modell eftersom han menar att

tangenten bara används av tummen, som dessutom använder mellanslagstangenten som viloplats och aldrig flyttas.

Modellen som Krzywinski har tagit fram innehåller många bra och motiverade ansträngningsvärden. Carpalx erbjuder därför en bra grundmodell att utveckla vidare på då den täcker beräkning av ansträning i händer och fingrar vid vanligt textskrivande. Krzywinski nämner dock själv vissa brister i Carpalx. Även om applikationen kan matas med vilket språk som helst och med vilka tecken som helst är representationen av shift-tangenterna något begränsad. Tangenter kan flyttas runt på tangentbordet utan restriktioner, men tecken som ligger i shift-läge (t ex ';' och ':') är alltid bundna till samma tangent. Carpalx kan inte ta fram en fullt optimal layout eftersom tecken som skrivs med nedtryckt shift-tangent inte kan flyttas runt oberoende av den tillägnade tangenten.

De tecken som ligger i altgr-läge faller under samma restriktioner som tecknen i shift-läge. Krzywinski betonar dock att just optimering för tangentbord i sammanhang där tecken i altgr-läge är vanligt förekommande kräver att Carpalx utvecklas ytterligare. Han ställer en öppen fråga på hemsidan för Carpalx, främst till länder som inte är engelskspråkiga och därför har andra tangent- och teckenuppsättningar, om tecken i shift-lägen ska kunna flyttas runt fritt på tangentbordet och om användandet av shift- och altgr-tangenterna ska tas med i beräkningarna av ansträngningsvärdet i modellen. Den underbyggande tanken är troligen att göra Carpalx så generell som möjligt och att en effektiv layout ska kunna tas fram oavsett träningsdata. Det är framförallt på grund av bristerna i beräkningen av shift- och altgr-tangenterna i Carpalx som det har väckts ett intresse hos oss att försöka förbättra modellen med funktionstangenterna i åtanke. I och med att modellen är så pass utförligt beskriven med såväl formler som testdata var den också ett självklart val att studera och utveckla utifrån.

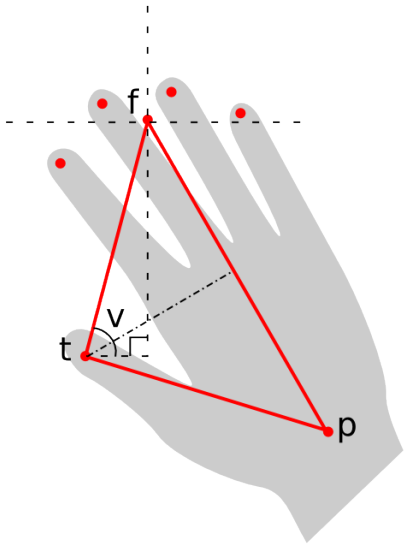
4.3 Implementation av betygsättningsmodell

Utifrån modellen i Carpalx byggdes en applikation som genererar en analys med avseende på en specificerad tangentbordslayout och vald testdata. Applikationen tar fram information om vilka tangenter som används mest och dess ansträngninsvärden. Utifrån denna data genereras bland annat värmekartor (7.1) vilka ger en bra bild om vilka tangenter som haft den högsta användningsfrekvensen och gör det förhållandevis lätt att upptäcka skillnader mellan layouterna. Applikationens modifierade betygsättningsmodell beskrivs i följande avsnitt.

4.3.1 Rotation effort

Som nämnt tidigare innehåller Java-kod en del tecken som är svåråtkomliga på tangentbordet. Måsvingarna och hakparenteserna är nämnda som sådana och eftersom

de är så vanligt förekommande i Java-kod är det motiverat att införa delar i modellen som beskriver ansträngningen för att skriva dessa tecken. Den viktigaste aspekten som rör användandet av altgr-tangenten är att tummen behöver flyttas från sin plats på mellanslagstangenten till altgr-tangenten som ligger relativt långt bort. När tummen byter plats placeras den direkt under handens övriga fingrar i en svag och något obekväm position. Denna komplexa rörelse kan beskrivas på många olika vis. Ett enkelt angreppssätt är att titta på den vinkel som bildas mellan tummen och de övriga fingrarna på handen. Eftersom tummen har en nästintill fast placering på tangentbordet, på mellanslagstangenten, kan denna vinkel användas för att mäta vinkeln på handleden. Figur 4.1 illustrerar förhållandet mellan tumme, fingrar och handled.



Figur 4.1. Rotation effort: approximation av handledsvinkel

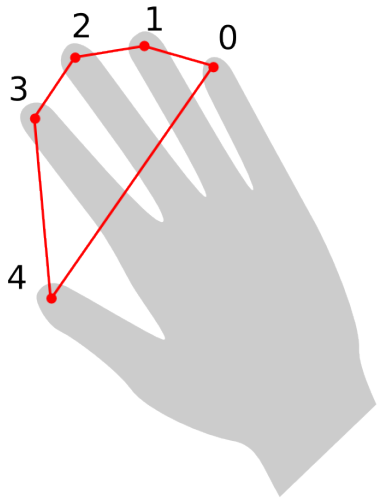
Den utritade triangeln i figuren är som synes likformig när handen är i avslappnat läge. Triangeln är framräknad från de tre punkterna f, t och p där f är ett medelvärde av de fyra fingrarnas positioner. Fingrarna har mycket rörliga leder vad gäller att greppa saker, men sidleds är deras rörelser betydligt mer begränsade vilket gör att triangeln håller sig någorlunda likformig när fingrarna flyttas runt. Vinkeln v ger inte ett direkt mått på vinkeln på handleden, men genom att använda v som en utgångspunkt kan den relativa vinkeln på handleden approximeras utifrån differensen $v_i - v_j$. En nackdel med angreppssättet är att linjen $t - f$ kommer ändra längd oberoende av linjen $t - p$ när fingrarna rör sig uppåt och neråt på tangentbordet och ge ett missvisande värde på vinkeln på handen. Eftersom handrörelsen för ett trigram kan se mycket annorlunda ut beroende på trigram valde vi att väga samman vinkelvärdet mellan tecknen på samma sätt som för base effort och penalty. Vinkeln som ges av differensen ger också information om åt vilket håll handen vrids. Detta

ger däremot information som är svår att väga in i det större sammanhanget varför vi valt att ta det absoluta värdet på differenserna. Låt oss kalla ekvationen för rotation effort som definieras enligt ekvation 4.7.

$$r_i = t_1|r_{i1}|(1 + t_2|r_{i1} - r_{i2}|(1 + t_3|r_{i2} - r_{i3}|)) \tag{4.7}$$

4.3.2 Penalty för fingrar

Den kostnad som Carpalx tilldelar varje finger ger intrycket av att vara mycket godtycklig i sina värden. Ansträngningen för ett finger är högst beroende av positionen för sina grannar. Att mäta avståndet mellan intilliggande fingrar erbjuder ett bättre sätt att betygsätta ett fingers position på tangentbordet. En sträckning för pekfingret mot den övre sifferraden räknas in i base effort för fingrets tillryggalagda sträcka, men den tar inte med i beräkningen hur långt ifrån de andra fingrarna det befinner sig. När tummen exempelvis ligger på altgr-tangenten för att skriva en vänster-måsvinge sträcks pekfingret mycket längre från tummen än om bara en enkel 7:a skrivs, då tummen vilar på mellanslagstangenten. Figur 4.2 illustrerar avstånden mellan de intilliggande fingrarna.



Figur 4.2. Finger penalty: avstånd mellan handens fingrar

Sträckningen beräknas som ett medelvärde mellan avstånden till de intilliggande fingrarna. Eftersom avstånden beräknas som en cirkel runt fingrarna behöver avstån-

den också indexeras cirkulärt. Eftersom antalet fingrar per hand är fem beräknar vi modulo 5 på indexeringen. Ekvation 4.8 visar förändringen.

$$P_{finger} = \frac{|d_{ij} - d_{ik}|}{2}, j \equiv_5 i - 1, k \equiv_5 i + 1 \tag{4.8}$$

4.3.3 Parametrar i modellen

Varje parameter i Carpalx är viktad med en variabel för att distribuera parametrarna efter ett visst förhållande. Krzywinski föreslår att vikta mer eller mindre oberoende parametrar jämnt så att alla parametrar bidrar lika mycket. Det totala ansträngningsvärdet e_i viktas därmed i Carpalx med förhållandet 1:1:1 vilket efter körningar på träningsdata ger ett ansträngningsvärde på 3.0. Base effort b_i och penalty p_i viktas efter förhållandet 60:30:10 vilket betyder att det första tecknet i trigrammet bidrar med 60%, det andra tecknet med 30% och det tredje tecknet med 10%. Nedan visas en tabell med vikter och förhållanden anpassade efter QWERTY-layouten med böckerna från litteraturlistan som indata.

Figur 4.3. Parametrar och vikter för carpalx[13]

$k_{\text{b,p,s}}$	$k_{1,2,3}$	w_0, $w_{\text{hand,row,finger}}$	P	f
0.3555	1	0, 1, 1.3088, 2.5948	$P_{\text{hand}} = 0, 0$	$f_{hand} = 1$
0.6423	0.367		$P_{\text{row}} = 1.5, 0.5, 0, 1$	$f_{row} = 0.3$
0.4268	0.235		$P_{\text{finger}} = 1, 0.5, 0, 0, 0, 0, 0.5, 1$	$f_{finger} = 0.3$

I vår utökade modell har vi lagt till parametrar för handrotation r motsvarande de för b, p och s. Parametrar för penalty för fingrar är samma som för P_{finger} då tillägget ersätter det gamla penalty-värdet för fingrarna. De inre vikterna inom trigramet $k_{1,2,3}$ för b och p delades upp i två separata viktmängder då tillägget av handrotation ändå krävde en uppsättning vikter oberoende av b och p. De nya vikterna benämns $k^{(b)}_{1,2,3}$, $k^{(p)}_{1,2,3}$ samt $k^{(r)}_{1,2,3}$.

4.4 Val av analysmaterial

Det bestämdes tidigt att Github[7] skulle användas för att hitta öppen källkod att analysera i kexjobbet. Github tillhandahåller versionshantering och öppnar därmed möjligheterna för den öppna källkodsvärlden vad gäller samarbete och skapande.

Eftersom programmeringsspåket Java är ett av de populäraste språken på Github valdes det som språk att undersöka närmre. Tre medelstora projekt bland de mest populära valdes som analysmaterial.

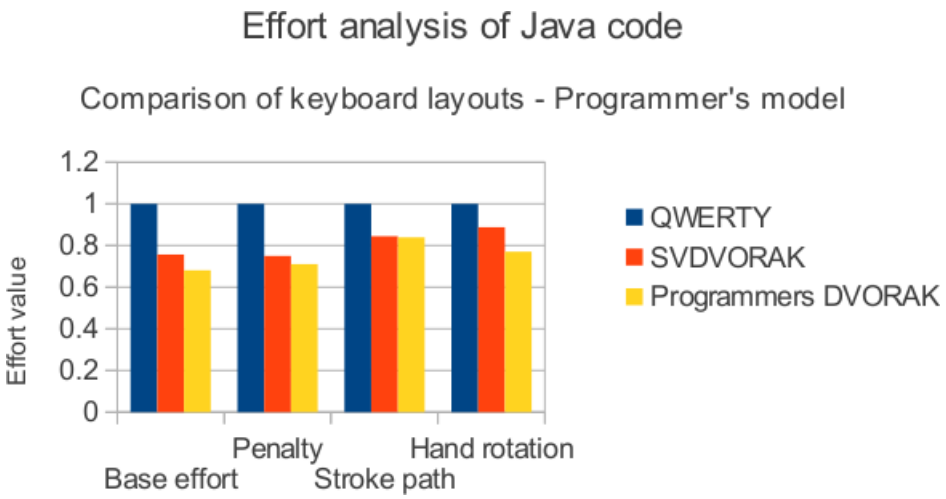
- Facebook Android SDK[8]
- Storm[9]
- JUnit[10]

Samtliga .java-filer som ingår i projekten analyseras med hjälp av programmet.

Kapitel 5

Resultat

Drygt fyra miljoner unicode-tecken fördelat på 928 .java-filer genomsöktes och analyserades med avseende på tre tangentbordslayouter. Figur 5.1 visar resultatet av analysen medan tabell 5.1 visar mer detaljerade värden från samma körning.



Figur 5.1. Triad-analys på Java-projekt

	Sum	Base effort	Penalty	Stroke path	Hand rotation
QWERTY	*4*	1	1	1	1
SVDVORAK	*3.24*	0.757	0.750	0.845	0.888
Programmers DVORAK	*3.002*	0.681	0.711	0.839	0.771

Tabell 5.1. Ansträngningsvärden för triad-analys på Java-projekt

Vi ser att SVDVORAK får ungefär 1/4 bättre resultat än QWERTY med våra mätresultat. Det visas också att programmers DVORAK får ungefär 1/3 bättre

resultat med samma mätresultat. Fördelningen mellan de olika ansträngningsvärdena är mycket likfördelade och ingen kan sägas få tydligt bättre resultat med någon layout. Tittar vi närmare på rotation effort (stapeln längst till höger) ser vi att resultatet blev stadigt bättre med såväl Programmer's DVORAK, eftersom den layouten har betydligt bättre fördelning specialtecken på tangentbordet. Det är dock fortfarande den ursprungliga modellen som ger tydligt bäst resultat med DVORAK-layouterna. Med detta skall däremot inte sägas att våra tillägg till modellen inte har bidragit med något till modellen eftersom resulatet ändå förbättras med de layouter som är bättre lämpade för programmeringskod. Tabell 5.2 visar de tjugo trigrammen med högst frekvens från analysen.

Trigram	Ansträngningsvärde	Antal förekomster
9, kommatecken, enter	7.758386872095469	25277
mellanslag, AltGr, 7	6.055428322055791	22198
I, O, N	3.7273523977545064	21625
AltGr, 7, enter	4.959401961060964	21184
AltGr, 0, enter	5.298162970858563	19002
vänster skift, 8, 9	5.195786104164604	18621
enter, AltGr, 0	19.45927719483591	17928
vänster skift, kommatecken, enter	7.008851678045026	17333
T, I, O	3.5621535894127647	16758
9, mellanslag, AltGr	5.7982202247662284	16231
I, C, mellanslag	2.2611146769174297	16139
L, I, C	0.9798721919711598	14659
mellanslag, T, H	2.523450069031023	14380
mellanslag, vänster skift, AS	3.6130393517041783	14113
mellanslag, vänster skift, 0	3.054814102278615	14081
vänster skift, 0, mellanslag	6.404741381863691	14049
I, N, G	2.165048165691544	13992
enter, mellanslag, mellanslag	8.561547978237657	13896
E, T, vänster skift	4.378081379730778	13667
S, T, R	2.796667135819024	13333
enter, mellanslag, vänster skift	9.77710821066671	13201
E, S, T	3.107478906367081	13104
kommatecken, enter, AltGr	9.77082420287202	12859
U, B, L	4.272580257238765	12571
vänster skift, S, T	5.0707912097548915	12217
B, L, I	2.781490687073814	12215
P, U, B	2.4136656512837895	12207
A, S, S	1.571813661797543	11956
vänster skift, 9, mellanslag	6.433859564628416	11816
E, N, T	2.8987182224595314	11715

Tabell 5.2. Trigraminformation

Se även de värmekartor som modellen genererade under Figurer och Tabeller.

Kapitel 6

Slutsatser

6.1 Jämförelserna

Upptäckten att QWERTY-layouten var den svagaste länken i kedjan var föga förvånande. Om man vill hitta en bättre layout än QWERTY, verkar det vettigt att göra en ansträngningsjämförelse över den domän text/kod/skrivet material man arbetar med.

Testerna i denna rapport visar på mindre ansträngningsvärde vid användning av någon av DVORAK-layouterna kontra QWERTY. Sett ur ett samhällsperspektiv skulle vi spara våra ansträngningar till någonting mycket bättre. Man skulle kunna tänka sig att de barn som växer upp nu och inte redan vant in sig med QWERTY, skulle tjäna på att lära sig en alternativ tangentbordslayout redan från början.

Vid en körning med litteraturlistan använd i Carpalx som indata fick vi däremot ett mindre sammanlagt ansträngningsvärde med QWERTY. Detta tyder alltså på att det är jobbigare att skriva kod än vanlig text.

En av de uppenbara nackdelarna med att göra en analys av källkod och att inte registrera knapptryck direkt vid kodande, är självklart att en hel del information går förlorad. Skriver man fel ofta måste man räkna in alla tryck på delete och backspace. Även tryck på förflyttningsknapparna och eventuella tangentbordsgenvägar går man miste om. Detta är dock något som inte trycker undan det resultat som presenterats i rapporten, eftersom det endast skulle utöka undersökningsrummet och inte avgränsa metodvalet.

6.2 Felkällor

Val av projekt som analysmaterial
De val som gjorts av projekt kan omöjligt representera all typ av Javakod som skrivits. När projekten valts har det gjorts med omsorg, men det innebär inte att man för den sakens skull kan uttala sig om hur tangentbordslayouterna skulle prestera utanför det rum av källkod som projekten spänner upp.

Albert Einstein lär ha sagt[11]:

> No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

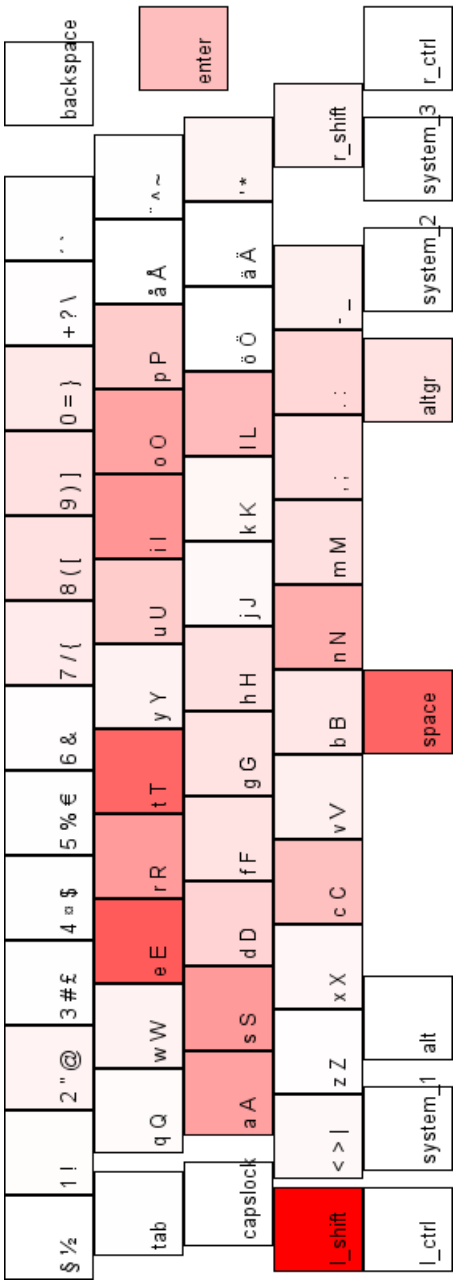
6.3 Avslutning

Som fortsatt arbete med rapporten kan man tänka sig att utöka tangentbordsanalysen till att inkludera fler modifikationstangenter och tangentbordslayouter. Applikationen vi byggt är inte bunden till några tangentbordslayouter eller modifikationstangenter i sig, och är därmed väl anpassad för en uppgradering.

En ytterligare utbyggnad skulle kunna vara en realtidstestning av layouterna. En sådan förbättring skulle fånga knapptryckningar som annars försvinner, exempelvis förflyttningar, redigeringskommandon och kodmiljökommandon. Detta var inledningsvis en av våra ambitioner att inkludera som en del av rapporten.

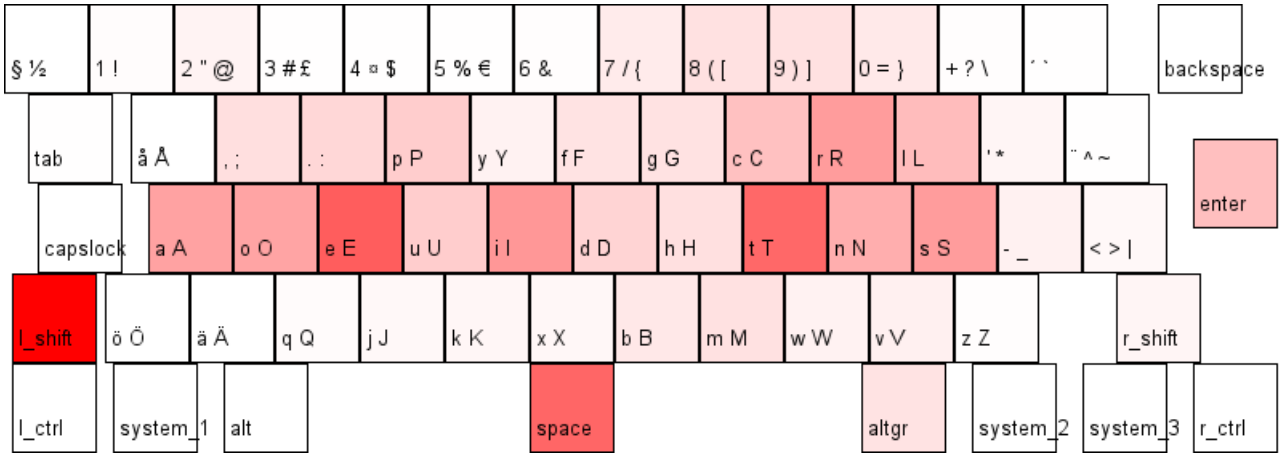
Kapitel 7

Figurer & Tabeller

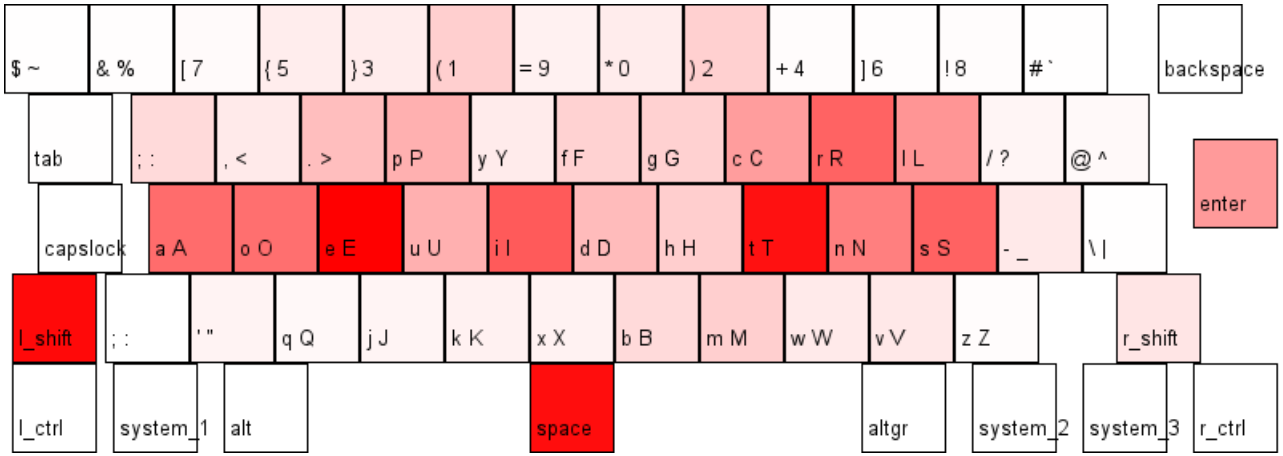


Figur 7.1. QWERTY-heatmap

Figur 7.2. SVDVORAK-heatmap



Figur 7.3. Programmer's DVORAK-heatmap



Figur 7.4. Parametrar och vikter för den modifierade modellen

$k_{\text{b,p,s,r}}$	$k^{(b)}_{1,2,3}$	$k^{(p)}_{1,2,3}$	$k^{(r)}_{1,2,3}$
1.666	0.207	0.229	0.0797
1.667	0.124	0.134	0.0408
0.625	0.0444	0.0490	0.0148
1.665			
w_0, $w_{\text{hand,row,finger}}$	P	f	
0, 0, 1.8834, 0.52483	$P_{\text{hand}} = 0, 0$	$f_{hand} = 1$	
	$P_{\text{row}} = 1.5, 0.5, 0, 1$	$f_{row} = 0.3$	
	$P_{\text{finger}} = 2, 1.5, 1, 1, 1, 1, 1, 1, 1.5, 2$	$f_{finger} = 0.3$	

Käll- och litteraturförteckning

[1] Sol Sherr, 1998, *Input devices*, United Kingdom Edition, London, Academic Press, INC. (LONDON) LTD.

[2] Greg Little, Robert C. Miller, March 2009, *Keyword programming in Java*, Automated Software Engineering, Volume 16, Issue 1, pp 37-71

[3] Layout med flera modifikationstangenter (obs, endast på tyska) `http://www.neo-layout.org/`

[4] Martin Krzywinski, *carpalx keyboard optimizer*, `http://mkweb.bcgsc.ca/carpalx/`

[5] *Harvard RSI Action*, april 2013, `http://www.rsi.deas.harvard.edu/what_is.html`

[6] *Fördjupning, Tangentbordet och pekdon*, Arbetsmiljöverket, april 2013 `http://www.av.se/teman/datorarbete/forebygg/datorn/fordjuptangentpek.aspx`

[7] *Github*, april 2013, `https://github.com`

[8] *Facebook Android SDK*, commit ed74ff363e3a7c32b54567e68cc7e86ea27bdbd7, april 2013, `https://github.com/facebook/facebook-android-sdk`

[9] *Storm*, A distributed realtime computation system, commit 61eee4da6533f3131dd0f875f620c5784baa816f, `https://github.com/nathanmarz/storm`

[10] *JUnit* Unittests for Java, commit beb1f4a80f7fa20523d40535fb81c1b8a7a9e638 `https://github.com/junit-team/junit`

[11] Genom Wikipedia, `http://en.wikipedia.org/wiki/Falsifiability#cite_ref-30`, april 2013, hittades följande källa:
Alice Calaprice, *The New Quotable Einstein*, 2005, USA: Princeton University Press and Hebrew University of Jerusalem. p. 291. ISBN 0-691-12074-9.

Calaprice påpekar att det inte är ett exakt citat av Einstein, utan snarare en tolkning och översättning av Einstein's "Induction and Deduction". Collected Papers of Albert Einstein Vol. 7, Document 28. The Berlin Years: Writings, 1918–1921. A. Einstein; M. Janssen, R. Schulmann, et al., eds.

[12] Svenska Akademins OrdBok, *KLAVIATUR*, april 2013, http://g3.spraakdata.gu.se/saob/show.phtml?filenr=1/120/63.html

[13] *carpalx model parameters*, april 2013, http://mkweb.bcgsc.ca/carpalx/?model_parameters

[14] Nationalencyklopedin, *Skrivmaskin*, april 2013, http://www.ne.se/lang/skrivmaskin

[15] Nationalencyklopedin, *Tangentbord*, april 2013, http://www.ne.se/lang/tangentbord

Kapitel 8

Bilagor

8.1 Bilaga A - Källkod

Keyboard

```
# coding: utf-8

import colorsys, math, re, sys

import matplotlib.pyplot as plt
import pygame

class Keyboard:
    unit_meter = 1.9
    keyboard_width = 15
    keyboard_height = 5

# The positions of the keys on a keyboard.
# The distance between two normal sized keys on the same row is equal to
# one. In this way the actual size of the keyboard can easily be tweaked
# only by measuring the same distance between two keys on a physical
# keyboard and multiply by every position.
    key_position = [
        [(0.0,0.0),(1.0,0.0),(2.0,0.0),(3.0,0.0),(4.0,0.0),(5.0,0.0),
         (6.0,0.0),(7.0,0.0),(8.0,0.0),(9.0,0.0),(10.0,0.0),(11.0,0.0),
         (12.0,0.0),(13.6,0.0)],
        [(0.3,1.0),(1.5,1.0),(2.5,1.0),(3.5,1.0),(4.5,1.0),(5.5,1.0),
         (6.5,1.0),(7.5,1.0),(8.5,1.0),(9.5,1.0),(10.5,1.0),(11.5,1.0),
         (12.5,1.0),(14.0,1.5)],
        [(0.4,2.0),(1.7,2.0),(2.7,2.0),(3.7,2.0),(4.7,2.0),(5.7,2.0),
         (6.7,2.0),(7.7,2.0),(8.7,2.0),(9.7,2.0),(10.7,2.0),(11.7,2.0),
         (12.7,2.0)],
        [(0.1,3.0),(1.2,3.0),(2.2,3.0),(3.2,3.0),(4.2,3.0),(5.2,3.0),
         (6.2,3.0),(7.2,3.0),(8.2,3.0),(9.2,3.0),(10.2,3.0),(11.2,3.0),
         (13.1,3.0)],
        [(0.1,4.0),(1.3,4.0),(2.6,4.0),(6.2,4.0),(10.1,4.0),(11.4,4.0),
         (12.7,4.0),(14.0,4.0)]
    ]

# PG = Paragraph, § ½
# PL = Plus, + ? \
```

```
# AC = Accent, ´ ‘
# AB = Angle brackets, < > |
# CF = Circumflex, ¨ ^ ~
# AS = Asterisk, ’ *
# LN = Line, - _
# CM = Comma, , ;
# DT = Dot, . :
# BS = Backspace
# EN = Enter
# TB = Tab
# CL = Capslock
# LS = Left shift
# RS = Right shift
# LC = Left ctrl
# RC = Right ctrl
# AL = Alt
# AG = Altgr
# SP = Space
# S1 - S3 = System keys
# N0 - N9 = Digits, 0 - 9
# A - Z AO AE OE = Characters, A - Z Å Ä Ö
    default_key_chars = {
        'AB':[u'<',u'>',u'|'], 'AC':[u'´',u'‘'], 'AS':[u'\'',u'*'],
        'CF':[u'¨',u'^',u'~'], 'CM':[u',',u';'], 'DT':[u'.',u':'],
        'LN':[u'-',u'_'],
        'PG':[u'§',u'½'], 'PL':[u'+',u'?',u'\\'],

        'BS':['backspace'], 'CL':['capslock'], 'EN':['\n'], 'TB':['\t'],
        'SP':[' '],
        'LS':['l_shift'], 'RS':['r_shift'], 'LC':['l_ctrl'], 'RC':['r_ctrl'],
        'AG':['altgr'], 'AL':['alt'],
        'S1':['system_1'], 'S2':['system_2'], 'S3':['system_3'],

        'N0':[u'0',u'=',u'}'],'N1':[u'1',u'!'],'N2':[u'2',u'"',u'@'],
        'N3':[u'3',u'#',u'£'],'N4':[u'4',u'¤',u'$'],'N5':[u'5',u'%',u''],
        'N6':[u'6',u'&'],'N7':[u'7',u'/',u'{'],'N8':[u'8',u'(',u'['],
        'N9':[u'9',u')',']'],

        'A':[u'a',u'A'], 'B':[u'b',u'B'], 'C':[u'c',u'C'], 'D':[u'd',u'D'],
        'E':[u'e',u'E'], 'F':[u'f',u'F'], 'G':[u'g',u'G'], 'H':[u'h',u'H'],
        'I':[u'i',u'I'], 'J':[u'j',u'J'], 'K':[u'k',u'K'], 'L':[u'l',u'L'],
        'M':[u'm',u'M'], 'N':[u'n',u'N'], 'O':[u'o',u'O'], 'P':[u'p',u'P'],
        'Q':[u'q',u'Q'], 'R':[u'r',u'R'], 'S':[u's',u'S'], 'T':[u't',u'T'],
        'U':[u'u',u'U'], 'V':[u'v',u'V'], 'W':[u'w',u'W'], 'X':[u'x',u'X'],
        'Y':[u'y',u'Y'], 'Z':[u'z',u'Z'],
        'AO':[u'å',u'Å'], 'AE':[u'ä',u'Ä'], 'OE':[u'ö',u'Ö']
    }
    programmers_dvorak_key_chars = {
        'AC':[u'´',u'¨'], 'AD':[u'&',u'%'], 'AP':[u'\'',u'"'], 'AT':[u'@',u'^'],
        'PS':[u'\\',u'|'], 'CM':[u',',u'<'], 'DL':[u'$',u'~'], 'DT':[u'.',u'>'],
        'FS':[u'/',u'?'], 'HS':[u'#',u'‘'], 'LN':[u'-',u'_'], 'SM':[u';',u':'],

        'BS':['backspace'], 'CL':['capslock'], 'EN':['\n'], 'TB':['\t'],
        'SP':[' '],
        'LS':['l_shift'], 'RS':['r_shift'], 'LC':['l_ctrl'], 'RC':['r_ctrl'],
        'AG':['altgr'], 'AL':['alt'],
        'S1':['system_1'], 'S2':['system_2'], 'S3':['system_3'],

        'N0':[u'*',u'0'],'N1':[u'(',u'1'],'N2':[u')',u'2'],
        'N3':[u'}',u'3'],'N4':[u'+',u'4'],'N5':[u'{',u'5'],
        'N6':[u']',u'6'],'N7':[u'[',u'7'],'N8':[u'!',u'8'],'N9':[u'=',u'9'],

        'A':[u'a',u'A'], 'B':[u'b',u'B'], 'C':[u'c',u'C'], 'D':[u'd',u'D'],
        'E':[u'e',u'E'], 'F':[u'f',u'F'], 'G':[u'g',u'G'], 'H':[u'h',u'H'],
        'I':[u'i',u'I'], 'J':[u'j',u'J'], 'K':[u'k',u'K'], 'L':[u'l',u'L'],
        'M':[u'm',u'M'], 'N':[u'n',u'N'], 'O':[u'o',u'O'], 'P':[u'p',u'P'],
        'Q':[u'q',u'Q'], 'R':[u'r',u'R'], 'S':[u's',u'S'], 'T':[u't',u'T'],
        'U':[u'u',u'U'], 'V':[u'v',u'V'], 'W':[u'w',u'W'], 'X':[u'x',u'X'],
        'Y':[u'y',u'Y'], 'Z':[u'z',u'Z'],
    }
```

```
# Descriptions of the keys
    key_description = {
        'AB':'< > |', 'AC':u'´ ‘', 'AG':'Altgr', 'AL':'Alt',
        'AS':'\' *', 'CF':u'¨ ^ ~', 'CM':', ;', 'DT':'. :',
        'LN':'- _', 'PG':u'§ ½', 'PL':'+ ? \\',

        'BS':'Back', 'CL':'Caps', 'EN':'Ret', 'TB':'Tab', 'SP':'Space',
        'LS':'LShift', 'RS':'RShift', 'LC':'LCtrl', 'RC':'RCtrl',
        'S1':'Sys1', 'S2':'Sys2', 'S3':'Sys3',

        'N0':'0 = }','N1':['1','!'],'N2':['2','"','@'],
        'N3':['3','#',u'£'],'N4':['4',u'¤','$'],'N5':['5','%',u''],
        'N6':['6','&'],'N7':['7','/','{'],'N8':['8','(','['],'N9':['9',')',']'],

        'A':['a','A'], 'B':['b','B'], 'C':['c','C'], 'D':['d','D'],
        'E':['e','E'], 'F':['f','F'], 'G':['g','G'], 'H':['h','H'],
        'I':['i','I'], 'J':['j','J'], 'K':['k','K'], 'L':['l','L'],
        'M':['m','M'], 'N':['n','N'], 'O':['o','O'], 'P':['p','P'],
        'Q':['q','Q'], 'R':['r','R'], 'S':['s','S'], 'T':['t','T'],
        'U':['u','U'], 'V':['v','V'], 'W':['w','W'], 'X':['x','X'],
        'Y':['y','Y'], 'Z':['z','Z'],
        'AO':[u'å',u'Å'], 'AE':[u'ä',u'Ä'], 'OE':[u'ö',u'Ö']
    }

# The qwerty keyboard layout.
    qwerty = [
        ['PG', 'N1', 'N2', 'N3', 'N4', 'N5', 'N6', 'N7', 'N8', 'N9',
         'N0', 'PL', 'AC', 'BS'],
        ['TB', 'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  'O',
         'P',  'AO', 'CF', 'EN'],
        ['CL', 'A',  'S',  'D',  'F',  'G',  'H',  'J',  'K',  'L',
         'OE', 'AE', 'AS'],
        ['LS', 'AB', 'Z',  'X',  'C',  'V',  'B',  'N',  'M',  'CM',
         'DT', 'LN', 'RS'],
        ['LC', 'S1', 'AL', 'SP', 'AG', 'S2', 'S3', 'RC']
    ]

# The svdvorak keyboard layout.
    svdvorak = [
        ['PG', 'N1', 'N2', 'N3', 'N4', 'N5', 'N6', 'N7', 'N8', 'N9',
         'N0', 'PL', 'AC', 'BS'],
        ['TB', 'AO', 'CM', 'DT', 'P',  'Y',  'F',  'G',  'C',  'R',
         'L',  'AS', 'CF', 'EN'],
        ['CL', 'A',  'O',  'E',  'U',  'I',  'D',  'H',  'T',  'N',
         'S',  'LN', 'AB'],
        ['LS', 'OE', 'AE', 'Q',  'J',  'K',  'X',  'B',  'M',  'W',
         'V',  'Z',  'RS'],
        ['LC', 'S1', 'AL', 'SP', 'AG', 'S2', 'S3', 'RC']
    ]

# The programmers dvorak keyboard layout.
    progdvorak = [
        ['DL', 'AD', 'N7', 'N5', 'N3', 'N1', 'N9', 'N0', 'N2', 'N4',
         'N6', 'N8', 'HS', 'BS'],
        ['TB', 'SM', 'CM', 'DT', 'P',  'Y',  'F',  'G',  'C',  'R',
         'L',  'FS', 'AT', 'EN'],
        ['CL', 'A',  'O',  'E',  'U',  'I',  'D',  'H',  'T',  'N',
         'S',  'LN', 'PS'],
        ['LS', 'SM', 'AP', 'Q',  'J',  'K',  'X',  'B',  'M',  'W',
         'V',  'Z',  'RS'],
        ['LC', 'S1', 'AL', 'SP', 'AG', 'S2', 'S3', 'RC']
    ]

# Fingers counted from left to right;
# Left: 0 - pinky, 1 - ring, 2 - middle, 3 - index, 4 - thumb
# Right: 5 - thumb, 6 - index, 7 - middle, 8 - ring, 9 - pinky
    finger_map = [
        [0,0,1,2,3,3,6,6,7,8,9,9,9],
        [0,0,1,2,3,3,6,6,7,8,9,9,9,9],
        [0,0,1,2,3,3,6,6,7,8,9,9,9],
        [0,0,0,1,2,3,3,6,6,7,8,9,9],
```

```
        [0,0,4,4,5,9,9,9]
    ]

# The resting position for each finger.
    finger_resting_position = [
        key_position[2][1], key_position[2][2],
        key_position[2][3], key_position[2][4],
        (key_position[4][3][0] - 1, key_position[4][3][1]),
        (key_position[4][3][0] + 1, key_position[4][3][1]),
        key_position[2][7], key_position[2][8],
        key_position[2][9], key_position[2][10]
    ]
# The order in which to calculate finger distance
    finger_distance_order = [
        [(0,1),(1,2),(2,3),(3,4),(4,0)],[(9,8),(8,7),(7,6),(6,5),(5,9)]]
# The finger positions for each finger.
    finger_position = []
# The distances between the fingers on each hand
    finger_distances = []
# The initial rotation of the hand
    hand_rotation = []
# Finger statistics stored as
# [distance traveled, key presses, latest distance traveled]
    finger_stats = []
# The currently pressed key
    current_key = None

    def __init__(self, keyboard):
        pygame.init()
        self.set_keyboard(keyboard)

"""
Sets a keyboard layout for this keyboard object.
"""
    def set_keyboard(self, keyboard):
        if keyboard == 'qwerty':
            self.keyboard = self.qwerty
            self.key_chars = self.default_key_chars
        elif keyboard == 'svdvorak':
            self.keyboard = self.svdvorak
            self.key_chars = self.default_key_chars
        elif keyboard == 'progdvorak':
            self.keyboard = self.progdvorak
            self.key_chars = self.programmers_dvorak_key_chars
        else:
            raise Exception('No such keyboard.')

        self.reset()

# recalculate finger distances and hand rotation
        self.finger_distances = self.calculate_finger_distances()
        self.hand_rotation = self.calculate_hand_rotation()

"""
Calculates the distance between two keys on the keyboard.
"""
    def calculate_key_distance(self, src, dst):
        return self.unit_meter * math.sqrt(math.pow(src[0] - dst[0], 2) \
                                           + math.pow(src[1] - dst[1], 2))

"""
Calculates distances between each finger on each hand.
"""
    def calculate_finger_distances(self):
        D = [[0 for i in range(0, 5)] for j in range(0, 2)]
        F = [[0 for i in range(0, 5)] for j in range(0, 2)]
# calculate distances between fingers
        for i in range(0, 2):
            for j, (k, l) in enumerate(self.finger_distance_order[i]):
                D[i][j] = self.calculate_key_distance(
                        self.finger_position[k],
```

```
                        self.finger_position[l])
# calculate the total distance for each finger
        for i in range(0, 2):
            for j, k in self.finger_distance_order[0]:
                F[i][k] = (D[i][j] + D[i][k]) / 2

        return F

"""
Returns the row and column index of the key on the keyboard.
"""
    def get_key_index(self, key):
        if hasattr(self, 'keyboard'):
            for r in range(0, len(self.keyboard)):
                for c in range(0, len(self.keyboard[r])):
                    if key == self.keyboard[r][c]:
                        return (r, c)
            return None
        else:
            raise Exception('Keyboard not defined')

"""
Returns the row, column and key index of the character on the keyboard.
"""
    def get_char_index(self, char):
        if hasattr(self, 'keyboard'):
            for r in range(0, len(self.keyboard)):
                for c in range(0, len(self.keyboard[r])):
                    if char in self.key_chars[self.keyboard[r][c]]:
                        return (r, c,
                                self.key_chars[self.keyboard[r][c]].index(char))
            return None
        else:
            raise Exception('Keyboard not defined')

"""
Returns the characters corresponding key.
"""
    def get_key(self, char):
        index = self.get_char_index(char)
        return self.keyboard[index[0]][index[1]]

"""
Returns the key position of the character on the keyboard.
"""
    def get_key_position(self, char):
        if isinstance(char, basestring):
            index = self.get_char_index(char)
        else:
            index = char

        pos = self.key_position[index[0]][index[1]]
        if isinstance(pos[0], float):
            return pos
        else:
# map the space bar to the left thumb
            return pos[0]

"""
Returns the modification key for the given character,
or None if none is pressed.
"""
    def get_modification_key(self, char):
        if isinstance(char, basestring):
            idx = self.get_char_index(char)
        else:
            idx = char
        finger = self.finger_map[idx[0]][idx[1]]

        if idx[2] == 1:
```

```
# shift was pressed
            if finger == 0:
# right shift was pressed if left pinky is occupied
                return 'RS'
            else:
# otherwise left shift was pressed
                return 'LS'
        elif idx[2] == 2:
# altgr was pressed
            return 'AG'

        return None

"""
Calculates the rotation of each hand for the current finger positioning.
"""
    def calculate_hand_rotation(self):
        rad2deg = 180.0 / math.pi

        fpos = self.finger_position

# calculate for left hand
        x = 0
        y = 0
        for p in fpos[0:4]:
            x += p[0]
            y += p[1]
        x = x / 4
        y = y / 4
        left = math.atan((y - fpos[4][1]) / (x - fpos[4][0])) * rad2deg
        if x > fpos[4][0]:
            left += 90

# calculate for right hand
# measure the
        x = 0
        y = 0
        for p in fpos[6:10]:
            x += p[0]
            y += p[1]
        x = x / 4
        y = y / 4
        right = math.atan((y - fpos[5][1]) / (x - fpos[5][0])) * rad2deg
        if x < fpos[5][0]:
            right -= 90

        return (left, -right)

"""
Returns the relative hand rotation to the initial finger positions.
"""
    def get_relative_hand_rotation(self):
        rot = self.calculate_hand_rotation()
# maintain a positive rotation for inwards hand motion
        return (rot[0] - self.hand_rotation[0], self.hand_rotation[1] - rot[1])

"""
Returns statistics for the last key stroke, given on the form:
['hand, 'row, 'finger, 'finger distance traveled', 'hand rotation',
'hand finger stretch']
"""
    def get_key_stroke_stats(self):
        idx = self.current_key

# set the currently used finger
        finger = self.finger_map[idx[0]][idx[1]]
# set the finger travel distance
        finger_dist = self.finger_stats[finger][2]
# set the currently used hand
        if finger < 5:
            hand = 0
```

```
        else:
            hand = 1
# set the current rotation of the used hand
        rotation = self.get_relative_hand_rotation()[hand]
# set the currently used row
        row = idx[0]
# set the finger stretch matrices
        finger_init = self.finger_distances
        finger_stretch = self.calculate_finger_distances()[hand]
        for i in range(0, 5):
            finger_stretch[i] = finger_stretch[i] - finger_init[hand][i]

        return [hand, row, finger, finger_dist, finger_stretch, rotation]

"""
Simulates a key press on the keyboard.
"""
    def press_key(self, key):
        index = self.get_key_index(key)
        if index:
# set the currently pressed key
            self.current_key = index

# fetch the finger mapped to the pressed key
            finger = self.finger_map[index[0]][index[1]]

# calculate the distance from the previously pressed key
# and the the currently pressed key
            d = self.calculate_key_distance(
                    self.finger_position[finger],
                    self.key_position[index[0]][index[1]])
            self.finger_stats[finger][0] += d
            self.finger_stats[finger][1] += 1
            self.finger_stats[finger][2] = d

# update finger position
            self.finger_position[finger] = self.key_position[index[0]][index[1]]

# update the heat map
            self.total_key_presses += 1
            self.heat_map[index[0]][index[1]] += 1
            if self.heat_map[index[0]][index[1]] > self.max_key_presses:
                self.max_key_presses = self.heat_map[index[0]][index[1]]
        else:
            raise Exception('Key not found.')

"""
Resets the current key analyzer.
"""
    def reset(self):
        self.reset_finger_positions()

        self.finger_stats = [[0, 0, 0] \
                             for i in range(0, len(self.finger_position))]

        self.total_key_presses = 0
        self.max_key_presses = 0
        self.heat_map = []
        for i in range(0, len(self.keyboard)):
            self.heat_map.append([0 for j in range(0, len(self.keyboard[i]))])

"""
Moves the each finger to their corresponding home positions.
"""
    def reset_finger_positions(self):
        self.finger_position = [pos for pos in self.finger_resting_position]

"""
Creates a keyboard heat map for the recorded recorded data and saves to file.
"""
    def save_heat_map_image(self, file_name):
```

```
        width = 800
        height = 280
        dx = width / self.keyboard_width
        dy = height / self.keyboard_height
        surface = pygame.Surface((width, height))
        surface.fill(pygame.Color(255,255,255))

        black = pygame.Color(0, 0, 0)
        red = pygame.Color(255, 0, 0)
        for r in range(0, len(self.keyboard)):
            for c in range(0, len(self.keyboard[r])):
                pos = self.key_position[r][c]
                if isinstance(pos[0], float):
                    pos = (pos[0] * dx, pos[1] * dy)
                else:
# place the space bar in the middle of the two thumb positions
                    pos = ((pos[0][0] + (pos[1][0] - pos[0][0]) / 2) * dx, pos[0][1] * dy)

# draw the key heat
                key_surface = pygame.Surface((dx, dy))
#key_surface.set_alpha(255 * self.heat_map[r][c] / self.max_key_presses)
                ratio = float(self.heat_map[r][c]) / self.max_key_presses
                rgb = colorsys.hsv_to_rgb(0, ratio, 1)
                color = pygame.Color(int(rgb[0] * 255), int(rgb[1] * 255), int(rgb[2] * 255))
                pygame.draw.rect(key_surface, color, (0, 0, dx, dy))
                surface.blit(key_surface, pos)

# draw the key border
                pygame.draw.rect(surface, black, (pos[0], pos[1], dx, dy), 1)

# draw the key label
                font = pygame.font.SysFont('Arial', 13)
                string = ' '.join(self.key_chars[self.keyboard[r][c]])
                if string == u'\n':
                    string = 'enter'
                elif string == u'\t':
                    string = 'tab'
                elif string == ' ':
                    string = 'space'
                label = font.render(string, 1, black)
                surface.blit(label, (pos[0] + 4, pos[1] + dx - 20))

        pygame.image.save(surface, file_name)

    def print_stats(self):
        total_key_presses = 0

        for s in self.finger_stats:
            total_key_presses += s[1]

        sorted_fingers = [(f, self.finger_stats[f]) for f in range(0, len(self.finger_stats))]
        sorted_fingers.sort(key=lambda f: f[1][1], reverse=True)

        print 'Finger Key presses Distance'

        for f in sorted_fingers:
            print '%d     %4.1f%%       %.2f m' % (f[0],
                100 * float(f[1][1]) / total_key_presses,
                self.unit_meter * f[1][0])
```

Triad Analyzer

```
import math
import os
```

```
import re
import sys
import time

from counter import Counter
from keyboard import Keyboard

# (b) base, (p) penalty, (s) path and (k) position weights
bpsk_w = [1.6656710479322678, 1.6667649927294148, 0.6250000000000017, 1.6647623825225086]

# weights for each character in the triad
trib_w = [0.20767107947891264, 0.1244003026841748, 0.044481603082494554]
trip_w = [0.22865600845267892, 0.13380439007749528, 0.049028095934152735]

# weights for each character penalty
p_w = [0, 0, 1.883353344658963, 0.5248343465233842]

# hand penalties [left, right]
p_hand = [0, 0]
# row penalties
p_row = [1.5, 0.5, 0, 1, 0]
# finger penalties
p_finger = [2.0, 1.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.5, 2.0]

# hand, row and finger path weights
s_w = [1.1801762314080677, 0.1247283095038429, 0.15454939605992649]

# hand rotation weights
r_w = [0.07969459062773394, 0.04080630720671516, 0.014758336278503998]

# hand rotation and finger stretch position weights
k_w = [1.0, 1.0]
# stretch weights for each character
t_w = [1.0, 1.0, 1.0]
# stretch weights for each space between fingers
tf_w = [1.0, 1.0, 1.0, 1.0, 1.0]

# set the keyboard
keyboard = 'qwerty'
kb = Keyboard(keyboard)

def main():
    print 'Calculating effort for %s keyboard...' % keyboard

    triad_means = []
    triad_occurences = []
    directory = sys.argv[1]
    file_type = sys.argv[2]
    directories = os.walk(directory)
    if re.match('win', sys.platform):
        slash = '\\'
    if re.match('linux', sys.platform):
        slash = '/'
    total_time = time.time()

    java_files = []
    for root, dirs, files in directories:
        java_files += [root+slash+f for f in files if f.endswith(file_type)]

    total_files = len(java_files)
    for i, f in enumerate(java_files):
        print 'Processing (%d/%d) %s...' % (i+1, total_files, f)

        file = open(f, 'r')
        string = file.read()
# remove multi line comments (ignoring newlines)
        string = re.compile(r'/\*.*?\*/', re.DOTALL).sub('', string)
# remove one line comments
        string = re.compile(r'//.*').sub('', string)
# remove tabs
```

```
        string = string.replace('    ', '')

        mean, occurence = run_optimizer(string)
        triad_means.append(mean)
        triad_occurences += occurence

# print time
        seconds = long(round(time.time() - total_time))
        minutes = int(math.floor(seconds / 60.0))
        seconds = int(seconds - minutes * 60)
        print '(total time: %d min %d sec)' % (minutes, seconds)

    mean = [
# effort sum
        [0, 0, 0, 0],
# base effort
        [0, 0, 0],
# penalty effort
        [0, 0, 0],
        [0, 0, 0, 0],
# path effort
        [0, 0, 0],
# position effort
        [0, 0, 0]] # rotation

    for m in triad_means:
        for i in range(0, len(m)):
            for j in range(0, len(m[i])):
                mean[i][j] += m[i][j]
    n = len(triad_means)
    for i in range(0, len(mean)):
        for j in range(0, len(mean[i])):
            mean[i][j] /= n

# count triad occurences
    sorted_triads = Counter(triad_occurences).most_common(200)

# write result to file
    fname = 'triadlog_' + str(long(round(time.time() * 1000))) + '.txt'
    output = open('./triadlogs/' + fname, 'w')
    output.write('Test results for %s keyboard on %s files in "%s".\n' % \
                 (keyboard, file_type, directory))
    output.write('Total files analyzed: %d\n' % total_files)
    seconds = long(round(time.time() - total_time))
    minutes = int(math.floor(seconds / 60.0))
    seconds = int(seconds - minutes * 60)
    output.write('Total time taken: %d min %d sec\n' % (minutes, seconds))
    print '\nTotal time taken: %d min %d sec' % (minutes, seconds)

    text = ['Effort sum:', 'Base:', 'Penalty:', 'Penalty contribution:',
            'Path:', 'Rotation:']
    output.write('\nCalculated effort: %f\n\n' % (sum(mean[0])))
    print 'Calculated effort: %f\n' % (sum(mean[0]))
    for i in range(0, len(mean)):
        output.write(str(text[i]) + '\n' + str(mean[i]) + '\n')
        print text[i]
        print mean[i]

    total_triads = len(triad_occurences)
    output.write('\nTriad occurences:\n')
    output.write('Total triads: %d\n' % total_triads)
    for t in sorted_triads:
        output.write(str(t) + '\n')

"""
Runs a triad analysis and displays the calculated effort.
"""
def run_analyzer(string):
# Calculate effort
    t = time.time()
```

```
    effort = analyze_triads(string, False)

    print 'Done in %f seconds' % (time.time() - t)
    print 'Effort: %f' % (sum(effort) / len(effort))

"""
Runs a triad analysis and shows the average of all contributing parameters.
"""
def run_optimizer(string):
# Analyze the string with trigrams
    timer = time.time()
    effort, triads = analyze_triads(string, True)

    mean = [
# effort sum
        [0, 0, 0, 0],
# base effort
        [0, 0, 0],
# penalty effort
        [0, 0, 0],
        [0, 0, 0, 0],
# path effort
        [0, 0, 0],
# position effort
        [0, 0, 0]] # rotation

    ratio = [
# effort sum
        [0, 0, 0, 0],
# base effort
        [0, 0, 0],
# penalty effort
        [0, 0, 0],
        [0, 0, 0, 0],
# path effort
        [0, 0, 0],
# position effort
        [0, 0, 0]] # rotation
    effort_sum = 0
    for e in effort:
        effort_sum += e[0]
# effort sum
        for i in range(0, len(mean[0])):
            mean[0][i] += e[1][i][0]
# base effort
        for i in range(0, len(mean[1])):
            mean[1][i] += e[1][0][1][i]
# penalty effort
        for i in range(0, len(mean[2])):
            mean[2][i] += e[1][1][1][0][i]
        for i in range(0, len(mean[3])):
            mean[3][i] += e[1][1][1][1][i]
# path effort
        for i in range(0, len(mean[4])):
            mean[4][i] += e[1][2][1][i]
# rotation effort
        for i in range(0, len(mean[5])):
            mean[5][i] += e[1][3][1][i]
# calculate the means
    size = len(effort)
    for i in range(0, len(mean)):
        for j in range(0, len(mean[i])):
            if isinstance(mean[i][j], (int, float)):
                mean[i][j] /= size
        m = max(mean[i])
        for j in range(0, len(mean[i])):
            if mean[i][j] > 0:
                ratio[i][j] = 1 / (mean[i][j] / m)

# order the triads
```

```
    ordered_triads = []
    for t in triads:
        for i in range(0, t[1]):
            ordered_triads.append((t[0], t[2][0]))#, t[1], t[2][0]))

    print '  Done in %.1f seconds. Effort: %f.' % (time.time() - timer,
                                                   effort_sum / len(effort))

    return (mean, ordered_triads)

"""
Analyzes a given triad for the current keyboard.
"""
def analyze_triads(string, verbose):
# character sequence length
    n = 3
    keys = []
    mod_key_prev = None

# add possible modification keys
    for char in list(string):
        idx = kb.get_char_index(char)
        if idx:
            mod_key = kb.get_modification_key(char)
            if mod_key and not mod_key == mod_key_prev:
                keys.append(mod_key)
            mod_key_prev = mod_key
            keys.append(kb.keyboard[idx[0]][idx[1]])

    effort = []
    triads = []
# for each triad
    for i in range(0, len(keys) - n + 1):
        triad = keys[i:i+n]
        string = ','.join(triad)
        found = False
        for t in range(0, len(triads)):
            if string in triads[t]:
                triads[t][1] += 1
                e = triads[t][2]
                found = True
                break
        if not found:
            e = calculate_effort(triad, verbose)
            triads.append([string, 1, e])
        effort.append(e)

    return (effort, triads)

"""
Calculates the effort for the given triad.
"""
def calculate_effort(triad, verbose):
    stats = []
# reset the keyboard statistics
    kb.reset()
    for key in triad:
# press the key and store the finger distance traveled
        kb.press_key(key)
        stats.append(kb.get_key_stroke_stats())

    b = base_effort(stats, verbose)
    p = penalty_effort(stats, verbose)
    s = path_effort(triad, stats, verbose)
    k = position_effort(stats, verbose)

    v_e = [bpsk_w[0] * b[0], bpsk_w[1] * p[0], bpsk_w[2] * s[0],
           bpsk_w[3] * k[0]]
    effort = sum(v_e)
```

```
    if verbose:
        return [effort, [[v_e[0], b[1]], [v_e[1], p[1]], [v_e[2], s[1]],
                         [v_e[3], k[1]]]]
    else:
        return effort

"""
Calculates the base effort given the triad statistics.
"""
def base_effort(stats, verbose):
    b = []
    for i in range(0, len(stats)):
# get the finger travel distance
        b.append(stats[i][3])

# calculate the base effort
    v_e = []
    v_e.append(trib_w[2] * b[2])
    v_e.append(trib_w[1] * b[1] * (1 + v_e[0]))
    v_e.append(trib_w[0] * b[0] * (1 + v_e[1]))
    v_e.reverse()
    effort = v_e[0]

    if verbose:
# show parameter contribution
        return [effort, v_e]
    else:
        return [effort]

"""
Calculates the penalty effort given the triad and its statistics.
"""
def penalty_effort(stats, verbose):
    p = []
    v_p = [0, 0, 0, 0]
    for i in range(0, len(stats)):
# calculate the penalty for each character
# NEW
        finger = stats[i][2]
        if stats[i][0] == 0:
            a_finger = finger
        else:
            a_finger = 9 - finger
# multiply by finger stretch
        p_e = [p_w[0], p_w[1] * p_hand[stats[i][0]],
               p_w[2] * p_row[stats[i][1]],
               p_w[3] * p_finger[finger] * stats[i][4][a_finger]]
# OLD
#p_e = [p_w[0], p_w[1] * p_hand[stats[i][0]],
# p_w[2] * p_row[stats[i][1]],
# p_w[3] * p_finger[stats[i][2]]]
        for j in range(0, len(p_e)):
            v_p[j] += p_e[j]
        p.append(sum(p_e))
    for i in range(0, len(v_p)):
        v_p[i] /= len(stats)

# calculate the penalty effort
    v_e = []
    v_e.append(trip_w[2] * p[2])
    v_e.append(trip_w[1] * p[1] * (1 + v_e[0]))
    v_e.append(trip_w[0] * p[0] * (1 + v_e[1]))
    v_e.reverse()
    effort = v_e[0]

    if verbose:
# show parameter contribution
        return [effort, [v_e, v_p]]
    else:
        return [effort]
```

```
"""
Calculates the path effort given the triad and its statistics.
"""
def path_effort(triad, stats, verbose):
    hand = path_hand_penalty(stats)
    row = path_row_penalty(stats)
    finger = path_finger_penalty(triad, stats)

# calculate the path effort
    v_e = [s_w[0] * hand, s_w[1] * row, s_w[2] * finger]
    effort = sum(v_e)

    if verbose:
        return [effort, v_e]
    else:
        return [effort]

"""
Calculates the path hand penalty ranging from 0 to 2:
0: both used, not alternating
1: alternating
2: same
"""
def path_hand_penalty(stats):
    h = [s[0] for s in stats]

    if (h[0] == h[1] != h[2]) or (h[0] != h[1] == h[2]):
        return 0
    if h[0] != h[1] != h[2]:
        return 1
    return 2

"""
Calculates the path row penalty.
"""
def path_row_penalty(stats):
    r = [s[1] for s in stats]

    if r[0] == r[1] == r[2]:
        return 0
    if r[0] <= r[1] <= r[2]:
        return 1
    if r[0] >= r[1] >= r[2]:
        return 2
    if r[0] == r[1] or r[1] == r[2] or r[0] == r[2]:
        return 3
    if r[0] < r[1] < r[2]:
        return 4
    if r[0] < 1 + r[1] or r[1] < 1 + r[2]:
        return 5
    if r[0] > r[1] > r[2]:
        return 6
    return 7

"""
Calculates the path finger penalty.
"""
def path_finger_penalty(triad, stats):
    h = [s[0] for s in stats]
    f = [s[2] for s in stats]

    if f[0] < f[1] < f[2] or f[0] > f[1] > f[2]:
        return 0
    if ((triad[0] == triad[1] and f[1] != f[2]) or
            (f[0] != f[1] and triad[1] == triad[2])):
        return 1
    if (h[0] == h[1] and f[0] != f[1]) or (h[1] == h[2] and f[1] != f[2]):
        return 2
    if f[0] > f[1] < f[2] or f[0] < f[1] > f[2]:
```

```
        return 3
    if f[0] != f[1] != f[2] and f[0] == f[2]:
        return 4
    if (f[0] == f[1] == f[2] and (triad[0] == triad[1] or
                                  triad[1] == triad[2] or
                                  triad[0] == triad[2])):
        return 5
    if ((f[0] == f[1] != f[2] or f[0] != f[1] == f[2]) and
            triad[0] != triad[1] != triad[2]):
        return 6
    return 7

"""
Calculates the path hand rotation penalty ranging from 0 to 3.
"""
def position_effort(stats, verbose):
#s = [s[5] for s in stats]
    r = [s[5] for s in stats]

# compute rotation effort
    v_r = []
    v_r.append(r_w[2] * abs(r[2]))
    v_r.append(r_w[1] * abs(r[1] - r[2]) * (1 + v_r[0]))
    v_r.append(r_w[0] * abs(r[0] - r[1]) * (1 + v_r[1]))
    v_r.reverse()
    effort = v_r[0]

    if verbose:
        return [effort, v_r]
    else:
        return [effort]

if __name__ == '__main__':
    main()
```