CHALMERS

Assisterad zonindelning

Zonindelningsförslag baserade på lastdata från entreprenadmaskiner

Examensarbete i Data och Informationsteknik

JONATHAN SARGENT
MARTIN CLAESSON

Instutitionen för Data och Informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2022
www.chalmers.se

Examensarbete 2022:24

Assisterad zonindelning

Zonindelningsförslag baserade på lastdata från entreprenadmaskiner

JONATHAN SARGENT
MARTIN CLAESSON

CHALMERS

Instutitionen för Data och Informationsteknik
Chalmers Tekniska Högskola
Göteborg, Sverige 2022

Assisterad zonindelning
Zonindelningsförslag baserade på lastdata från entreprenadmaskiner
JONATHAN SARGENT
MARTIN CLAESSON

Supervisor: Mathias Andreasson, CPAC
Examiner: Lars Svensson, Instutitionen för Data och Informationsteknik

Examensarbete 2022:24
Instutitionen för Data och Informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Sverige Telephone +46 (0)31 772 1000

Instutitionen för Data och Informationsteknik
Göteborg, Sverige 2022

Assisterad zonindelning
Zonindelningsförslag baserade på lastdata från entreprenadmaskiner
JONATHAN SARGENT
MARTIN CLAESSON
Instutitionen för Data och Informationsteknik
Chalmers Tekniska Högskola

Examensarbete

Sammanfattning

Stenbrotts storlek och förändringshastighet är båda faktorer för hur resurskrävande det är för bolag att hålla kartor uppdaterade.

I projektet används information genererat från existerande system i dumprar och hjullastare i en algoritm med 2 uppgifter, att kategorisera informationen och generera zoner baserat på de olika grupperna av noder.

Vilket möjliggörs genom tillgång till geografisk, temporal information, samt information angående vad maskinen gjorde. Målet är att en potentiell integration till existerande system, som sedan skulle kunna vara hjälpmedel för maskin-förare.

Genom problemdomänen definierar rapporten mål för en algoritm som ska kunna användas i produktion och vara till störst användning för maskinarbetare möjligt.

En initial design har i samarbete med produktägaren använts som utgångspunkt och en mvp har utvecklats och raffinerats.

Algoritmen har utvecklats genom agilt arbete med inspiration från scrum. Först skapades en simpel version som sedan förbättrades och reviderades efter bedömning från både utvecklare och produktägare. Genom användandet av biblioteket förenklar projektet utvärdering av de olika iterationerna.

Som resultat erhåller projektet en algoritm som utöver att klara av prestandakraven genererar beskrivningar av information som kan vara till hjälp vid gruvdrift. För vidare utveckling rekommenderas undersökande av alternativ som exempelvis gruppering av noder genom maskinlärning. Ytterliggare förbättringar skulle kunna involvera förmåga för algoritmen att utgå från existerande resulterande zoner för att hoppa över delar av arbetet och därmed ytterligare optimeras i prestanda.

algoritmer, agilt arbete, zoner, automatisering, stenbrott.

Abstract

Quarries are enormous and ever changing places of work. Because of that, drivers of wheel-loaders and dumpers benefit from the usage of maps to guide them during their work. However, keeping maps, along with descriptions of zones for different kinds of work up-to-date is expensive both when it comes to time and money.

In the thesis, the process of creating an algorithm dealing with zone-generation is described. The idea is based on reusing already generated data from vehicles in production. Reusage is possible because of the generation of a location in tandem with other data being created when using previously aforementioned vehicles.

An initial design is found through assistance from the product owner. The design later on gets implemented, iterated upon and evaluated through the usage of a plotting-framework.

The end result is an algorithm deemed to have potential both in terms of performance and precision.

Keywords: algorithm, agile, automation, quarry

Förord

Vi vill tacka Mathias Andreasson på CPAC, för det stöd och den motivation som han bidragit med som våran handledare under examensarbetets gång. Sen vill vi även tacka Annika Wånghed Muskantor, Director of HR & Communications på CPAC, som gav oss möjligheten att utföra detta examensarbetet på CPAC.

Jonathan Sargent, Martin Claesson

Göteborg, Juni 2022

Innehållsförteckning

Begrepp

Nod	En geografisk punkt där en händelse av intresse för arbetet inträffat
Zon	Ett område där en definierad typ av arbete ska utföras
Arbetsområde	Ett område som används för att lasta och dumpa olika material. I de flesta fallen är det ett dagbrott.
Java	Java är ett objektorienterat högnivåspråk.
JVM	”Java Virtual Machine”, är en virtuell miljö som Java-applikationer exekveras i.

1

Inledning

1.1 Bakgrund

Aktiva dagbrott är enorma operationer. Ett exempel på dagbrott är Aitikgruvan. Aitikgruvan är 3000 meter lång, 1000 meter bred och 450 meter djup, vilket gör det till det största dagbrottet för kopparutvinning i Skandinavien [1]. Utöver Aitikgruvan finns i Sverige 11 andra gruvor. Den genomsnittliga produktionen för en gruva i Sverige är 7.35 miljoner ton malm [2].
Malm utvinns genom bearbetning av själva miljön arbetet utförs i vilket leder till en hastigt föränderlig arbetsmiljö. förändringar av miljön ger upphov till att platser för arbetsuppgifter också ändras.
Inom gruvdrift används ofta dumprar för att samla upp utvunnet material på hjullastare för transport till efterföljande steg i någon utvinningsprocess. Storleken och den konstant föränderliga miljön ger upphov till ett behov av kartor för att förare till entreprenadmaskiner lättare ska kunna navigera sig.

CPAC har en produkt som de kallar för ”Volvo Co-Pilot”, som är ett system som integreras i entreprenadmaskiner så som hjullastare, grävmaskiner, dumprar med mera [3]. Detta systemet är uppkopplat till deras molntjänst Rock2Road där maskinerna rapporterar in allmän driftinfo så som exempelvis lastnings- och avlastningshändelser [4]. I Rock2Road kan användaren sedan dela upp arbetsområdet i zoner för att specificera vart laster och material ska ta vägen, vilket sedan visas i entreprenadmaskinernas integrerade system.

Gruvors storlek och förändringshastighet är båda faktorer för hur resurskrävande det är för bolag att garantera tillgång till uppdaterade kartor för förare till dessa entreprenadfordon. I ett, enligt CPAC, medelstort stenbrott med 2-3 hjullastare och cirka 10 dumprar, där varje fordon skapar en datapunkt vid lastning samt avlastning, genereras det totalt mellan $50 - 100$ tusen datapunkter under ett års arbete [4]. Även om användaren inte behöver hantera dessa enskilt, så visar detta än dock att området förändras kontinuerligt. Genom att till viss del automatisera hanteringen av zonerna, så lyfts den bördan från användaren, som i sin tur kan lägga sina resurser på annat arbete.

1.2 Syfte och mål

Det arbete som görs är ett försök till att till den grad möjligt inom de tidsramar som angetts, samla data som beskriver logistiken material genomgår från att den bryts till att den skickas och kartlägga samt uppdatera den kartläggningen över tid. Denna kartläggning ska hjälpa förare av entreprenadfordon att navigera sig under tiden de arbetar.

Ett framgångsrikt resultat skulle innebära att fordon skulle ha tillgång till stöd från kartor med logistisk information över deras arbetsuppgifter. Utan att någon manuellt behövt rita upp dem. Kartläggningen skall innehålla zoner för olika arbetsuppgifter. Dessa zoner kommer behöva vara uppdelade efter arbetsuppgift. Dessa zoner behöver även vara geometriskt utformade på ett sätt som är pedagogiskt utformat för en slutanvändaren. De resulterade zonerna bör inte vara alltför komplext utformade för slutanvändaren då det skulle kunna orsaka förvirring under navigation, dem bör inte heller vara alltför förenklade, då dem då skulle riskera att utelämna viktigt information till den.

Generering av zoner skall också vara möjligt att göras för den mängder data som kommer vara tillgänglig i verkligheten, eftersom den är i ordning 10 000. Kommer det finnas krav på att komplexiteten skall vara tillräckligt låg för att det skall vara generering skall vara ett möjligt alternativ.

1.3 Avgränsningar

Resultatet av projektet kommer vara en mjukvara utvecklad i kotlin, som kommer utvecklas helt fristående från andra system.

1.3.1 Mjukvara

Den resulterande mjukvaran kommer bestå av en algoritm som är kapabel att kategorisera och gruppera datapunkter och en algoritm som kan formge zonerna. Mjukvara för att representera dem kommer endast skapas i den mån som det kan tänkas assistera i skapandet av algoritmer som bättre uppfyller de mål som satts ut. Arbetet kommer inte presentera någon integration till ett produktionssystem.

Inom målen som eftersträvas med algoritmen inkluderas formen av resulterande zon. Detta för att det antas att en zon som består av mindre komplexa former är mer pedagogisk. En sådan undersökning hade sträckt sig långt utanför ramen av vad arbetet har tänkt att undersöka.

1.3.2 Prestandakrav

Eftersom det jobb en potentiell algoritm skulle utföra i en produktionsmiljö endast är någonting som skulle behöva utföras med dagar eller veckors periodicitet, finns ej anledning att ta prestanda i hänsyn i någon större mån än vad som görs i övrig, icke-prestandafokuserande mjukvara. Prestandan kommer däremot analyseras för

att säkerställa att mjukvaran är användbar.

2

Teknisk Bakgrund

2.1 Kotlin

Kotlin är ett multiparadigm-programspråk. Det är skapat för att användas i JVM och på Android-baserade enheter. Det har har stöd för objektorienterad, funktionell och procedurell kod. Det är interoperabelt med Java-kod vilket i sin tur gör att det går att använda existerande Java-bibliotek i koden[5].

2.2 JSON

JavaScript Object Notation (JSON), är protokoll för kommunikation som ofta används vid kommunikation mellan separata tjänster. En `Json`-fil skrivs och lagras i text-format. I en `Json`-fil lagras ett objekt alternativt en array som i sin tur är kapabla att innehålla de olika typerna definerade i `Json`-standarden: Number, String och Boolean samt rekursivt andra objekt eller arrayer. [6] [7].

2.3 Lets-Plot for Kotlin

Lets-Plot for Kotlin är ett Kotlin-bibliotek för att visualisera statistisk data. Detta kan göras exempelvis genom diagram eller kartor [8]. I detta projekt används det för att visualisera den data som algoritmen skapar.

2.4 Java Swing

Java Swing är ett enkelt och väldokumenterat Java bibliotek för att skapa grafiska användargränssnitt. Det innehåller bland annat fördefinierade moduler för att skapa nya fönster, knappar och för att strukturera upp samtliga moduler i gränssnittet [9].

2.5 Agilt arbete och Scrum

Att arbeta agilt innebär att skapa produkter iterativt, med en kortare planeringshorisont, att dela in sitt arbete i små iterationer ger möjlighet att mer kontinuerligt utvärdera sin produkt och därefter kunna anpassa sig efter resultat från utvärderingen. Detta gör bland annat att mindre tid kan spenderas i arbete som inte kommer

användas i slutprodukten.
Scrum är en av flera existerande agila ramverk som kan användas. I Scrum delas arbete upp i tidsintervaller som kallas sprintar, inför en sprint planeras förväntat arbete som ska avklaras. Efter sprintar utvärderas både det arbete som gjorts och arbetsmetodiken för att hitta potentiella förbättringar. [10].

3
Metod

Utvecklandet av projektet har skett iterativt med inslag av arbetsmetodiken scrum. Under projektets gång har dagliga avstämningsmöten tagit plats i slutet av dagen samt ett veckomöte med produktägaren. Tillgång till kontinuerliga uppdateringar av framsteg och motgångar gav möjlighet att naturligt använda sig av den vetenskapliga metoden, alltså att med små steg testa nya saker och allteftersom utvärdera dem.

3.1 Verktyg

För projektet har programmeringsspråket Kotlin använts, detta för att kotlin tillåter användare att modellera sina problem både funktionellt, där funktioner är förstaklass konstruktioner och det därmed är tillåtet att skicka funktioner som argument till andra funktioner, och spara dem som värden. Det är även objektorienterat, där kotlin har ekvivalenta konstruktioner till Java, med klasser, interfaces, abstrakta klasser. Att kotlin är mångsidigt gör att det finns liten risk att ha problem att modellera problemen som kan tänkas uppstå under projektets gång.

För versionshantering har git använts.

3.2 Minimum viable product (MVP)

En MVP togs fram i samråd med produktägaren som hade kunskapen att kunna formulera problemen så att vi hade möjlighet att formulera produkten utifrån det. Det som bestämdes var att algoritmen skulle klara av följande:

- Skapa zoner med enbart en typ av material.
- Läsa in data i JSON format.
- Lågpolygona zongränser, exempelvis rektanglar.
- Skilja på zoner med samma material men som inte var nära varandra.

3.3 Testning

För att avgöra huruvuda lösningen som togs fram planerar vi att skriva automatiska tester för att kunna mäta prestanda. Vi kommer även manuellt granska resulterande zoner med hjälp av ett grafiskt gränssnitt. Kriterier som kommer spelar in vid

granskning kommer vara:

- Zonens form.
- Att den inte tar med noder som är för långt bort.
- Hur den följer nodernas placering.
- Hurvida zonen omsluter samtliga noder.
- Hur mycket extra yta zonen har.

3.4 Testdata

Vid testning av algoritmen användes två typer av data:

- Verklig data från produktägaren
- Slumpgenererad data

Där den av produktägaren tillhandahållna datan primärt användes för att utvärdera zonernas uppdelning och utformning, medans den slumpgenererade datan även användes för att utvärdera prestandan av algoritmen.

4

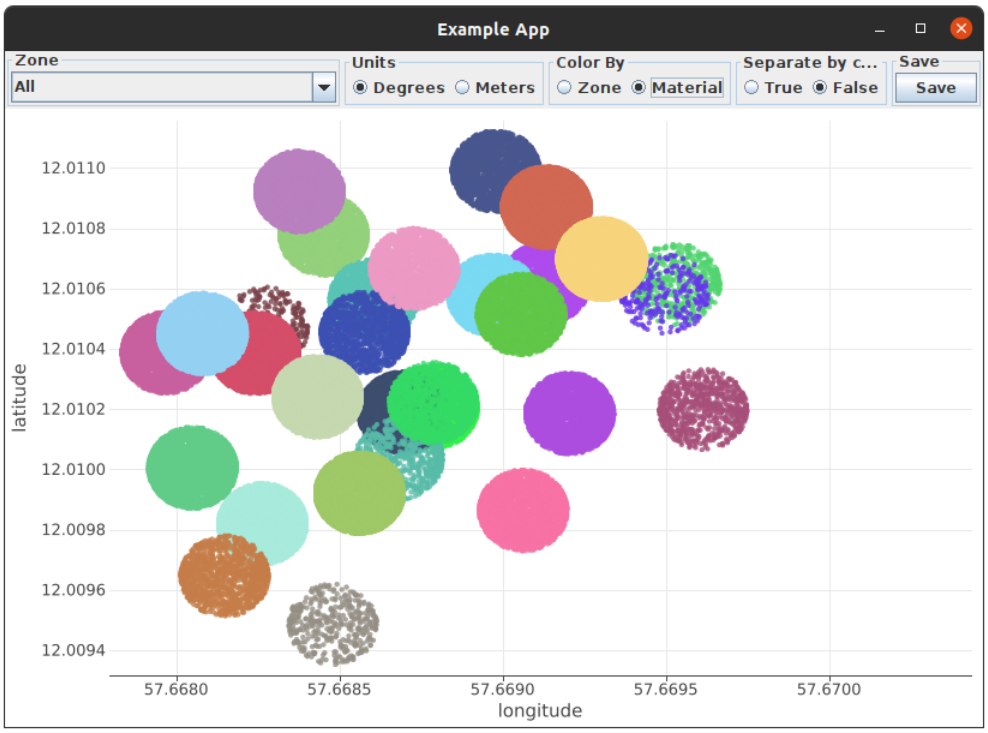
Genomförande

Projektet inleddes med att identifiera produktägarens behov och krav, samt att förstå hur algoritmerna skulle användas. Genom detta framkom det att det i första hand inte är tänkt att lösningen ska vara helautomatisk, utan snarare att slutanvändaren ska ha en möjlighet att vid behov kunna köra algoritmen och få förslag på hur zoner i området borde se ut, Vilket i sin tur gav att algoritmen inte kommer behöva vara tidskritisk.

I följande avsnitt kommer tillvägagångssättet att presenteras i större detalj.

4.1 Gränssnitt

För att enklare kunna tolka den data och det resultat som skapats, så skapades det först ett enkelt gränssnitt i Kotlin med en kombination av *Java swing* och *Lets-Plot*.



Figur 4.1: Projektets gränssnitt med slumpade datapunkter färgade efter dess material

I gränssnittet finns inställningar för att påverka hur resultatet presenteras, exempelvis att kunna visa endast en zon, färga efter zon eller material och till sist om olika typer av händelser ska ha olika former i grafen.

4.2 Datamodeller

För att kunna hantera den data som läses in ska kunna hanteras i algoritmerna så behövde Datamodeller skapas som reflekterar datan. Nedan följer en övergripande beskrivning av datamodellerna.

4.2.1 BucketLoad

Modellen BucketLoad (se A.2) representerar data en grävmaskin alternativt hjullastare skulle ha skickat in till systemet.

4.2.2 HaulCycle

Modellen Haulcycle (se A.1) representerar data en dumper eller lastbil skulle ha skickat in till systemet.

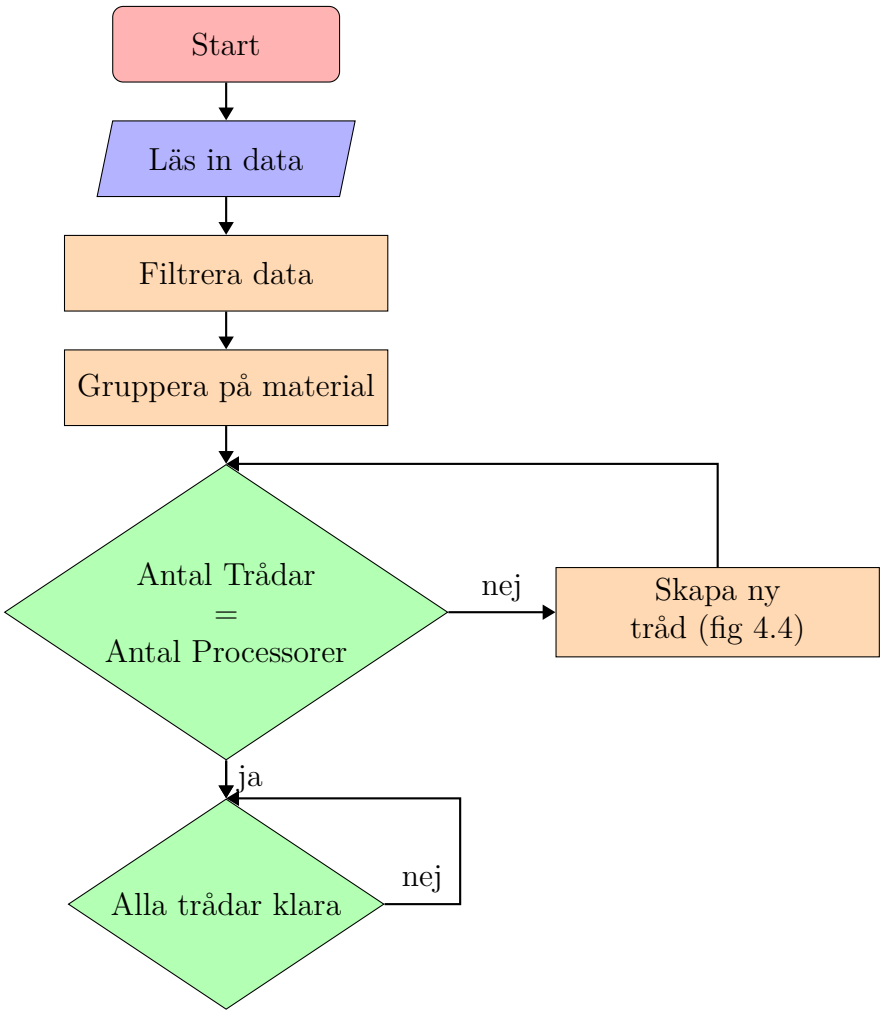
4.2.3 Order

Modellen Order är en abstrakt klass som skapar en gemensam typ för BucketLoad och HaulCycle eftersom de två modellerna skiljer sig åt i strukturen. Med hjälp utav Order så kan metoder som ska hantera BucketLoad och HaulCycle istället ta emot en Order och därmed inte behöva hantera skillnaderna i modellerna, Framförallt då det mesta av datan i respektive modell är överflödig med avseende på hur datan ska kategoriseras. De fält som är intressanta som denna modell exponerar är:

- Upphämtnings- och avlastningszoner
- Upphämtnings- och avlastningskoordinater
- Start- och stoptid
- Material

4.3 Design av algoritmen

Figur 4.2 representerar en övergripande design av algoritmen. Algoritmen använder sig av köer från javas standardbibliotek En kö är en datastruktur som är kapabelt att ta in ett värde, som den sätter längst bak genom opperationen `push()` som i javas fall kallas `add()` och att returnera samt ta bort det element som finns längst fram i kön genom metoden `pop()` som i javas fall kallas `poll()`. Eftersom applikationen är fler-trådad kräver det att datastruktuen är vad som brukar kallas för *thread-safe*. Vilket innebär att flera trådar ska kunna manipulera datastrukturen utan att korrumpera datan. Den specifika implementationen av kö som användes var javas *ConcurrentLinkedQueue*. Zonerna representerar dem under algoritmens processerande zonerna som håller på att skapas, som delade in noderna baserat på material och tidpunkt där noden skapats.



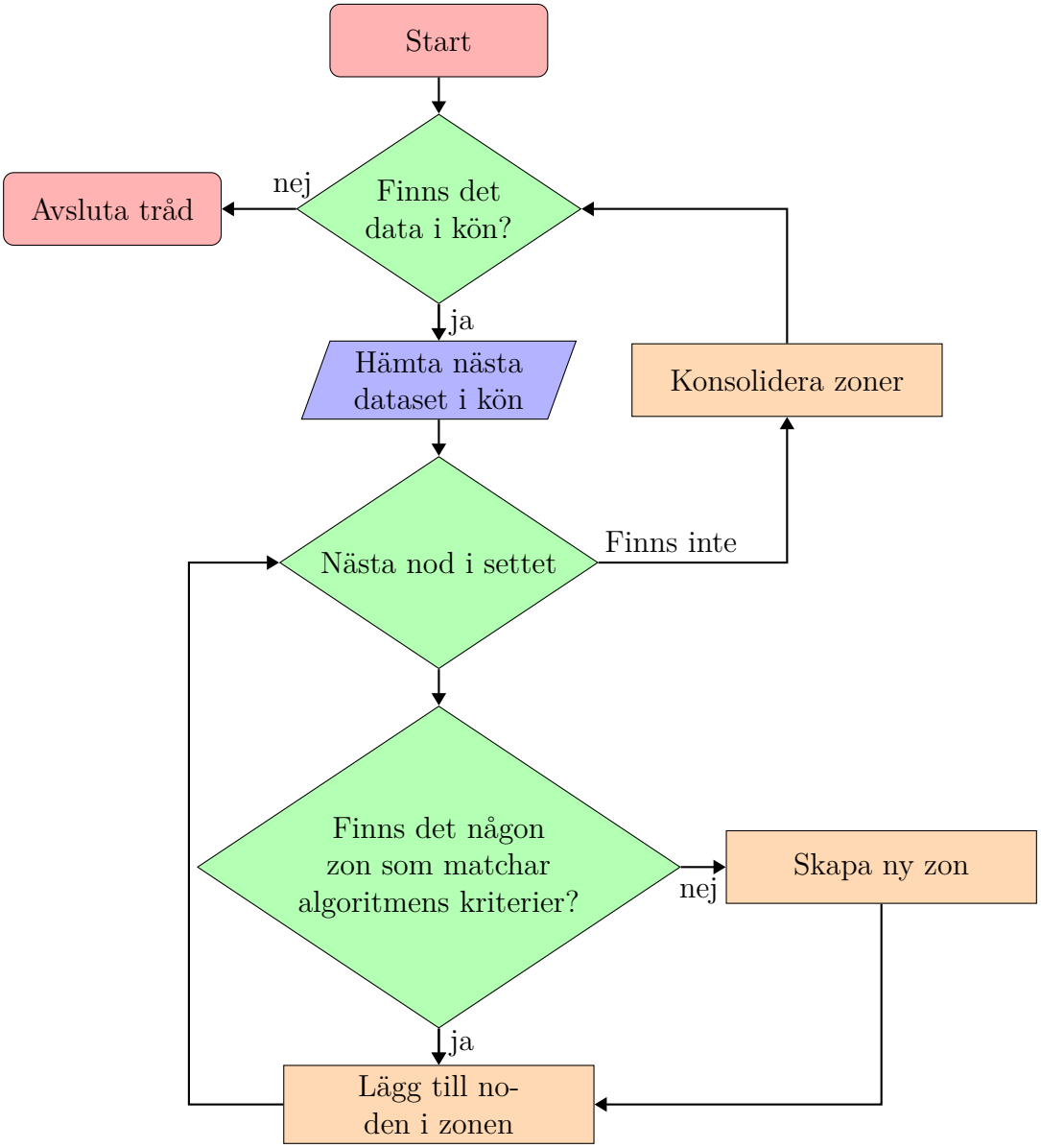
Figur 4.2: Algoritmens övergripande design

Algoritmen ska först och främst filtrera den datan som den givits för att ta bort noder som matchar någon av följande punkter:

- Äldre än den satta maxåldern.

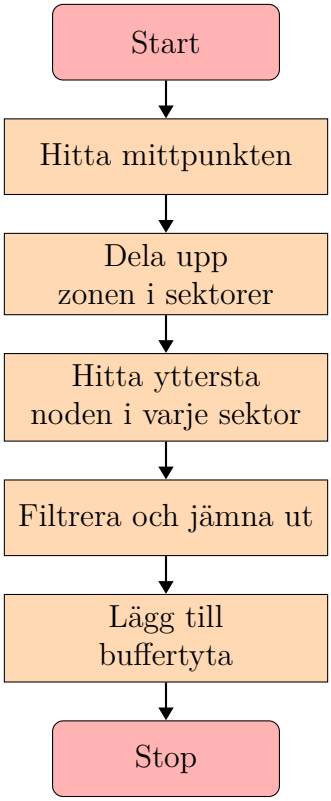
- Ligger långt utanför området som alla andra noder finns i.
- Materialet finns i listan av ignorerade material.

Därefter ska en enkel gruppering av noderna göras baserat på deras material, innan algoritmen skapar så många arbetartrådar den rimligtvis kan för att hantera datan, och vänta tills dessa är färdiga.



Figur 4.3: Algoritmtrådens övergripande design

Algoritmens arbetartrådar har en gemensam kö som innehåller de tidigare skapade grupperingarna av noder, tråden hämtar nästa tillgängliga dataset och går systematiskt (baserat på deras plats i föregående datastruktur) genom noderna och utför en jämförelse mellan noden och redan existerande zoner. Om ingen zon matcher kriterierna så skapas en ny zon och noden läggs sedan till i zonen. När alla noder har hanterats så görs en jämförelse mellan alla zoner för att avgöra om det finns zoner som kan konsolideras till större zoner. Kriterier för att två zoner ska kunna konsolidera med varandra är att dem består utav samma material och att dem befinner sig nära nog varandra, gränsen för hur nära zoner kan vara varandra utan att konsolideras bestäms innan körning av algoritmen. Tråden upprepar sedan detta mönster tills kön är tom, varpå den stänger ner sig själv.



Figur 4.4: Zonalgoritmens övergripande design

4.4 Skapandet av algoritmen

Grundpelaren och projektets huvuddel var algoritmen som skulle ta en indata på en samling noder och med hjälp av dessa härleda hur eventuella zoner skulle kunna se ut i området.

4.4.1 Gruppering av noder

Att dela på material baserade sig på att zoner i verkligheten tenderar att vara uppdelade på just material [4].

För att grupperna noderna användes vilken typ av material noden bestod av, samt en maxdistans en nod fick ha till nästa nod i samma zon, för att fortfarande räknas som en nod i den zonen. Att bestämma ett för lågt värde innebär att zoner som i verkligheten var avsedda för ett arbete, nu enligt algoritmen var fler zoner. Ett för högt värde skulle innebära att zoner som i verkligheten var områden för olika jobb, nu representerats som ett område.

För den datan som fanns att tillgå blev det bästa värdet på avstånd 5 meter. Detta gav en zonindelning som mest liknade det personal tenderade att dela in stenbrotten själva.

För att sedan hitta vilka noder som hör ihop så skapas en lista där varje nod blir en egen zon. Sedan jämför algoritmen samtliga zoner mot varandra i följande steg:

- Hämta den nod från båda zonerna som är närmst den andra zonens geografiska mittpunkt.
- Jämför avståndet mellan dessa, om avståndet är mindre än gränsvärdet så absorberar den första zonen den andra, och den andra tas bort från listan.
- Efter att zonen jämförts med alla andra zoner i listan sker ett av följande scenarion:
  - Om zonen förändrats läggs den till i listan på nytt.
  - Om zonen är oförändrad tas den bort från kön och algoritmen går vidare.

Detta upprepas tills listan är tom varpå algoritmen sparar de kvarvarande zonerna och går vidare till nästa materialgrupp.

4.4.2 Hitta zonernas kanter

Nästa problem för algoritmen var att skapa zonernas kanter. För detta behövde algoritmen först hitta vilka noder som ligger ytterst i zonen. Detta gjordes genom att dela upp zonen i 360 sektorer. Där vilken sektor en nod tillhör kunde avgöras genom dess vinkel till zonens geografiska mittpunkt. För att sedan skapa kanter hittades den nod med längst avstånd till zonens geografiska mittpunkt för varje sektor. För att undvika att få för stora variationer i kanterna ignoreras datanoder som är närmare mitten än genomsnittet för alla kantnoder i zonen. Kanten definerade sedan som en polygon vars linjers ändar utgjordes av dessa noder. Efter detta läggs det sedan till ett valbart avstånd som kanten flyttas ut, vilket skapar en buffertyta för framtida tillväxt.

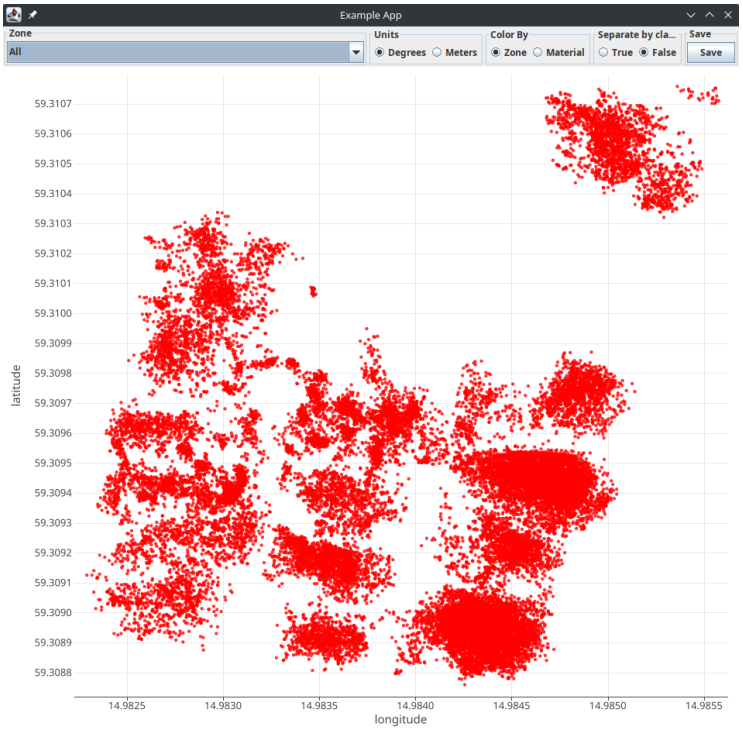
5

Resultat

Projektet har uppnått de mål som var uppsatta, produkten är en välpresterande algoritm som genererar zoner på ett förutsägbart och upprepningsbart sätt.

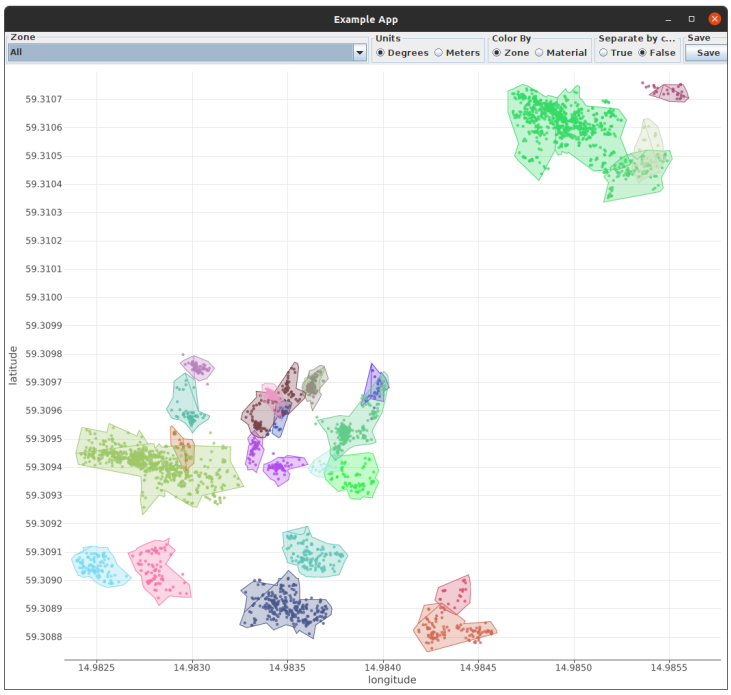
5.1 Slutprodukt

Nedan följer hur algoritmen kan ta in data (Figur 5.1) och få ut zoner anpassade till innehållet (Figur 5.2)



Figur 5.1: Visualisering av indatan innan hantering

I Figur 5.1 visas hur testdatan ser ut innan algoritmen har körts på den, notera här hur det finns ett antal ensamma noder som ligger utanför det område som majoriteten av noderna befinner sig i.



Figur 5.2: Visualisering av de av algoritmen skapade zoner, olika färger representerar olika material

Här i Figur 5.2 visas de resulterande zonerna som algoritmen har skapat, efter att ha filtrerat bort oönskade material, gamla datapunkter och noder som ligger utanför området.

5.2 Prestanda

Även om bedömningen gjordes att algoritmen inte behövde vara tidskritisk, så är fortfarande prestanda något som produktägaren ser som meriterande. Efter ett flertal iterationer av algoritmen har dess exekveringstid nått en nivå som ligger över förväntningarna.

I tabell 5.1 och 5.2 ges den tid algoritmen tar för en satt datamängd, på en dator med sex tillgängliga processorkärnor med 12 processortrådar. Ur dessa tabeller kan det utläsas att algoritmen kan förväntas ta 35-200 ms för den förväntade datamängden, givet liknande förhållanden. Algoritmen delar upp det arbete som behöver göras baserat på datanodernas material, där fler tillgängliga trådar skulle kunna innebära bättre prestanda. Bäst prestanda nås när antalet material är minst lika stort som antalet tillgängliga processortrådar, vilket går att härleda genom Figur 5.3. Det ska dock tilläggas att exekveringstiderna kan fluktuera något om datorn gjort något i bakgrunden samtidigt.

Med grafen i figur 5.3 kan det urskönjas att algoritmen har en bra tidskomplexitet, som för den förväntade datamängden ligger någonstans mellan $O(n \cdot \log(n))$ och $O(n)$, Där n är antalet datanoder.

Antal noder	Exekveringstid (ms)
1000	2.9426
2000	2.7966
4000	3.2438
8000	6.1623
16000	14.4137
32000	35.7089
64000	87.2834
128000	202.0521
256000	465.2266
512000	1055.9299

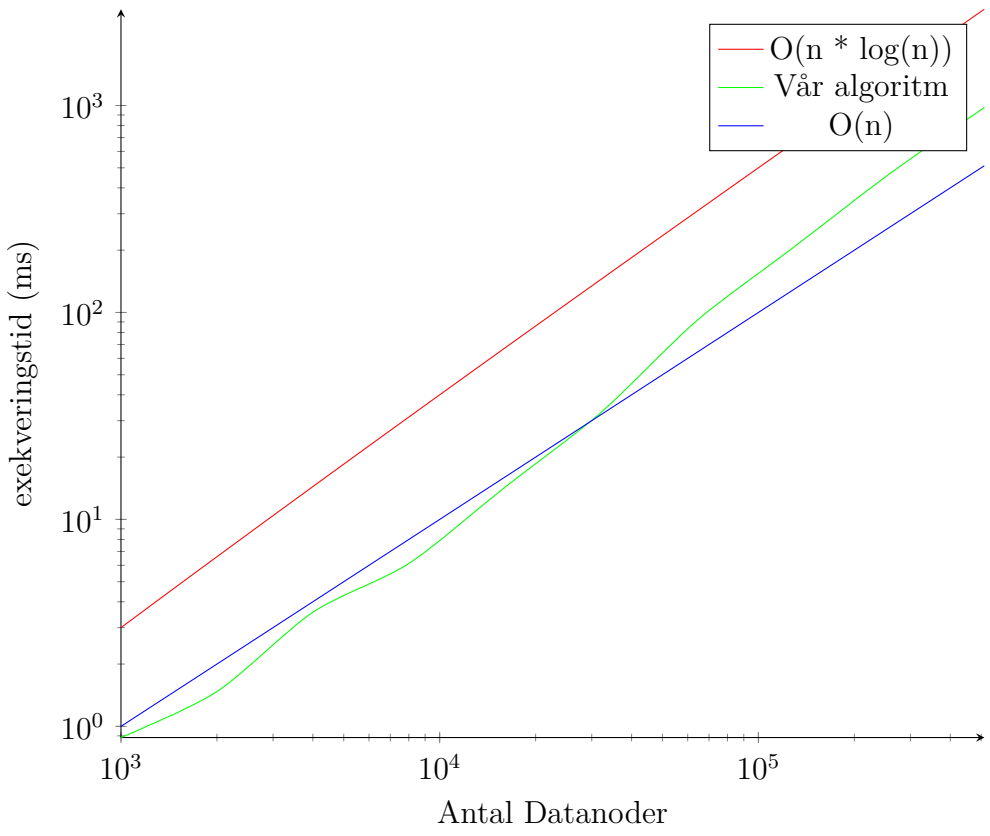
Tabell 5.1: Genomsnittlig Exekveringstid över 100 omgångar, för givet antal noder slumpmässigt fördelat på 30 olika material

Antal noder	Exekveringstid (ms)
1000	0.8830
2000	1.4723
4000	3.5590
8000	6.1230
16000	14.2817
32000	32.7578
64000	91.1275
128000	204.6539
256000	463.5440
512000	979.3040

Tabell 5.2: Genomsnittlig Exekveringstid över 100 omgångar, för givet antal noder jämnt fördelat på 30 olika material

Antal material	Exekveringstid (ms)
1	1866.9577
2	1379.7674
3	1294.2593
4	1178.0704
5	1123.0826
6	1097.5664
7	1089.9865
8	1042.9691
9	1030.2886
10	1038.5721
11	1023.9767
12	1012.5782
13	1012.4920
14	1040.0452
15	1044.8828
16	1055.5676
17	1023.1316
18	1017.2628
19	1012.4180
20	1015.6459

Tabell 5.3: Genomsnittlig Exekveringstid över 100 omgångar, för 512 000 noder jämnt fördelat på givet antal olika material



Figur 5.3: Illustration av tidskomplexitet, baserat på tabell 5.2
.

6

Diskussion

6.1 Resultat och Analys

Det har varit väldigt intressant och givande att se produkten utvecklas genom varje iteration, en skillnad som stod ut extra mycket var när vi parallelliserade algoritmen.

Valet att direkt skapa någon form av grafiskt gränssnitt för att bedöma hur väl algoritmen presterar, var definitivt rätt, det ökade produktiviteten med en stor marginal, då resultatet av varje förändring snabbt kunde visualiseras.

Däremot behöver algoritmen nog utvecklas ytterligare innan den kan sättas i bruk i verkligheten, exempelvis genom att förbättra hur algoritmen bygger upp zonerna. detta och mer kommer täckas i kapitel 7.

6.2 Miljö och Etik

Rapporten önskar att skapa en produkt vars önskade och förväntade utfall kommer minimera utsläpp av fordon. Resultatet, alltså en produkt som ger en förbättrad möjlighet för förare att korrekt navigera sina fordon bedöms därför vara etisk.

Gruvdrift är förövrigt en essentiell industri för människans nuvarande levnadssätt, vi känner inte till metoder som kan att ersätta den på ett mer miljövänligt sätt. Således leder valet att avstå från optimering av (fortfarande existerande) gruvdrift endast till mer utsläpp. Vidare bedöms arbetet att inte strida mot någon av stadgarna listade i Sveriges ingenjörers hederskodex [11].

6.3 Reflektion

Den plan som sattes ut för projektet visade sig vara väl anpassad för arbetet, genom att följa den agila arbetsmetoden samt en testdriven utveckling blev det lätt att styra projektet framåt.

Det gränssnittet som först utvecklades för att visualisera de av algoritmen genererade zonerna, visade sig vara ett väldigt hjälpsamt verktyg som vid flera tillfällen hjälpte till att identifiera och fixa problem som uppstod.

Den av uppdragsgivaren tillhandahållna testdatan var väldigt givande, då det gav en referens till hur slutanvändaren kan tänkas lägga zonerna. I efterhand hade det

däremot kunnat vara bra för projektet om flera olika scenarion hade funnits tillgängliga.

6.3.1 Arbetsmetodik

Den iterativa arbetsmetoden gav upphov till att det blev enklare att spåra projektets framsteg. Problem som uppstod upplevdes aldrig riskera projektets tidsplan på ett sätt som inte gick att anpassa projektet till.

Det upplevdes också som att handledare och Produktägare Mathias hade bra nog insyn i projektet för att vi skulle kunna hålla en bra dialog om nuvarande utmaningar och uppgifter vilket var till stor hjälp.

Någonting som hade kunnat vara bättre vore dokumentationen av vår kod, eftersom konsekvenserna från Covid-19 ledde till att slutförandet av arbetet försenades tvingades vi att lägga ett klart projekt på is under en längre tid. Om vi hade dokumenterat koden bättre tror vi det hade varit en kortare process att göra en slutpresentation av arbetet.

7

Förslag till fortsatt arbete

Den resulterande algoritmen är långt ifrån perfekt och det finns stor förbättringspotential för att ytterligare utöka dess effektivitet, precision eller funktionalitet. Några punkter som dykt upp under arbetets gång presenteras nedan.

7.1 Identifiera zontyper

I produktägarens system finns primärt tre olika typer av zoner: lagringszon, lastzon och grävzon. Som kan identifieras efter följande kriterier:

- Grävzon: Enbart upphämtnings-händelser i zonen
- Lastzon: Ofta en blandning av material som hanteras i zonen, händelser varierar mellan upphämtning och avlastning.
- Lagringszon: Oftast ett material men kan vara flera, händelser varierar mellan upphämtning och avlastning.

Genom att låta algoritmen identifiera vilken typ zonerna är så besparar detta ytterligare tid åt slutkunden.

7.2 Zoner med flera material

I vissa fall händer det att zoner kan innehålla blandade material, i nuläget hanterar inte algoritmen detta utan skapar i så fall två överlappande zoner. En möjlig lösning för att kunna hantera detta hade kunnat vara att ge algoritmen en lista på material som kan blandas om deras zoner överlappar med en viss marginal, alternativt kolla på vilken typ av zon det är som hanteras.

7.3 Zonernas yttre kanter

Algoritmen hanterar i dagsläget formen på zonerna och deras yttre kanter på ett väldigt simplistiskt sätt. Medans den i de flesta fallen fungerar utmärkt och att den får med majoriteten av sina noder där igenom, så finns det fortfarande mycket förbättringspotential här. Exempelvis tror vi att ”Alpha Shapes” hade passat utmärkt för att lösa detta problem bättre[12], detta var däremot något som upptäcktes i slutet av projektet.

7.4 Utgå från redan existerande zoner

Ett sätt att vidare förbättra prestandan av algoritmen hade kunnat vara att utgå från redan existerande zoner, detta skulle kunna spara mycket beräkningskraft. Detta då en del av noderna skulle komma förgrupperade.

7.5 Förutse framtida förändringar

Utöver att förbättra prestanda eller precision hos algoritmen hade möjligheten att, baserat på hur zonerna har växt historiskt sett, förutse hur zonerna kommer växa och ta del av den informationen när algoritmen formar zonerna, kunnat minska hur ofta slutanvändaren behöver köra algoritmen eller justera zonerna.

7.6 Utnyttja historisk data

Genom att utnyttja historisk data skulle det gå att utesluta datanoder i en punkt, exempelvis om det har plockats upp lika mycket material i en punkt som det tidigare lastats av. Algoritmen hade i så fall kunnat ta bort dessa datanoder innan zonen skapats, vilket skulle kunna ge en zon mer stämmer överens med hur det ser ut i verkligheten.

Referenser

[1] *Aitikgruvan*, Nationalencyklopedin. URL: https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/aitikgruvan (hämtad 7 juni 2022).

[2] *Mineralstatistik*, Sveriges Geologiska undersökning, juli 2021. URL: https://www.sgu.se/mineralnaring/mineralstatistik/# (hämtad 7 juni 2022).

[3] *Construction - Volvo CO-Pilot*, CPAC. URL: https://cpacsystems.se/en/construction/ (hämtad 9 juni 2022).

[4] M. Andreasson, private communication, CPAC, maj 2022.

[5] Kotlin. "Kotlin FAQ," JetBrains, URL: https://kotlinlang.org/docs/faq.html (hämtad 2 juni 2022).

[6] *Introducing JSON*. URL: https://www.json.org/ (hämtad 1 juni 2022).

[7] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, mars 2014. DOI: 10.17487/RFC7159. URL: https://www.rfc-editor.org/info/rfc7159.

[8] Lets-Plot, *Open-source plotting library for statistical data*, JetBrains. URL: https://github.com/JetBrains/lets-plot (hämtad 1 juni 2022).

[9] *A swing architecture overview*, Oracle. URL: https://www.oracle.com/java/technologies/a-swing-architecture.html (hämtad 6 juni 2022).

[10] T. Stobierski, *Agile vs. Scrum: What's the Difference? | Northeastern University*, mars 2021. URL: https://www.northeastern.edu/graduate/blog/agile-vs-scrum (hämtad 8 juni 2022).

[11] *Sveriges ingenjörers hederskodex*, Sveriges ingenjörer. URL: https://www.sverigesingenjorer.se/om-forbundet/organisation/hederskodex/ (hämtad 9 juni 2022).

[12] K. Fischer, *Introduction to Alpha Shapes*, Stanford. URL: https://graphics.stanford.edu/courses/cs268-11-spring/handouts/AlphaShapes/as_fisher.pdf (hämtad 8 juni 2022).

A

Dataformat


```
{
  "siteId": 0,
  "vid": "Machine 14",
  "cycleStart": "2022-01-01T00:00:00.000Z",
  "bucketCount": 0,
  "loadedTravelDuration": 0,
  "cycleEnd": "2022-01-01T00:00:00.000Z",
  "payloadPercent": 100,
  "payloadKg": 40000,
  "carrybackPercent": 0,
  "carrybackKg": 0,
  "loadingLatitude": 12.12345678910111 ,
  "loadingLongitude": 12.12345678910111 ,
  "loadingAltitude": -0.875,
  "unloadingLatitude": 12.12345678910111 ,
  "unloadingLongitude": 12.12345678910111 ,
  "unloadingAltitude": 0.75,
  "zoneUUID": "",
  "zoneName": "",
  "materialName": "Waste",
  "geographicalLoadZoneName": "",
  "geographicalUnloadZoneName": "",
  "loadingStartedTimestamp": "2022-01-01T00:00:00.000Z",
  "geographicalLoadZoneUUID": "",
  "geographicalUnloadZoneUUID": "",
  "unloadZoneUUID": "",
  "unloadZoneName": "",
  "loadZoneUUID": "",
  "loadZoneName": ""
}
```


Figur A.1: Exempel på haul_cycle.json, Från fraktande maskiner.

```
{
  "siteId": 0,
  "timestamp": "2022-01-01T00:00:00.000Z",
  "chassisId": "Machine 13",
  "bucketId": 0,
  "payload": 1000,
  "carryTime": 344520,
  "pickUpPosition": {
    "longitude": 21.12345678910111,
    "latitude": 21.12345678910111,
    "height": 47.125,
    "rms": 0
  },
  "dumpPosition": {
    "longitude": 12.12345678910111,
    "latitude":  12.12345678910111,
    "height": 47.5,
    "rms": 0
  },
  "materialName": "Waste",
  "targetName": "Target",
  "cycleTime": 355316,
  "cycleDistance": 106,
  "operator": "",
  "pickUpZone": {
    "name": "zone1",
    "uuid": "1234-1234-1234-1234"
  },
  "dumpZone": {
    "name": "zone2",
    "uuid": "1234-1234-1234-1234"
  },
  "materialNameId": "",
  "workOrderId": 0,
  "targetId": 0
}
```


Figur A.2: Exempel på bucket_load.json, Från lastande maskiner.

B

Prestandatabeller

I detta appendix finns en uppsättning testresultat som körts på en 6 kärnig processor med 12 trådar.

Antal noder	Exekveringstid (ms)
1000	5.191462
2000	4.868463
4000	6.202353
8000	10.457112
16000	23.590781
32000	56.081004
64000	152.801014
128000	379.721412
256000	833.207448
512000	1919.575376

Tabell B.1: Genomsnittlig Exekveringstid över 20 omgångar, för givet antal noder jämnt fördelat på 1 olika material

Antal noder	Exekveringstid (ms)
1000	1.496365
2000	2.637164
4000	3.78455
8000	8.056616
16000	16.663812
32000	42.246842
64000	114.64293
128000	254.687569
256000	614.485464
512000	1378.121321

Tabell B.2: Genomsnittlig Exekveringstid över 20 omgångar, för givet antal noder jämnt fördelat på 2 olika material

Antal noder	Exekveringstid (ms)
1000	1.534874
2000	1.813726
4000	3.665872
8000	6.34218
16000	13.45281
32000	34.992578
64000	86.86466
128000	215.529367
256000	476.630566
512000	1113.724417

Tabell B.3: Genomsnittlig Exekveringstid över 20 omgångar, för givet antal noder jämnt fördelat på 6 olika material

Antal noder	Exekveringstid (ms)
1000	1.423141
2000	2.012127
4000	3.549159
8000	6.165767
16000	13.386521
32000	33.47816
64000	85.368187
128000	207.028103
256000	452.917973
512000	994.371535

Tabell B.4: Genomsnittlig Exekveringstid över 20 omgångar, för givet antal noder jämnt fördelat på 12 olika material

Antal noder	Exekveringstid (ms)
1000	1.187385
2000	1.422783
4000	2.631779
8000	6.634241
16000	14.592292
32000	31.60271
64000	97.544579
128000	194.190294
256000	455.029513
512000	1038.038414

Tabell B.5: Genomsnittlig Exekveringstid över 20 omgångar, för givet antal noder jämnt fördelat på 16 olika material