CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Ett inbäddat domänspecifikt språk för visualisering av datastrukturer och algoritmer

Ett modulärt verktyg för att illustrera datastrukturers beteende

Kandidatarbete vid institutionen för Data- och Informationsteknik

Eddie Zell
Hedi Kelesh
Zakariya Hassan
Mandus Högberg
Christian Mattsson
Joakim Torstensson

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sweden 2025

Kandidatarbete 2025

Ett inbäddat domänspecifikt språk för visualisering av datastrukturer och algoritmer

Ett modulärt verktyg för att illustrera datastrukturers beteende

Eddie Zell
Hedi Kelesh
Zakariya Hassan
Mandus Högberg
Joakim Torstensson
Christian Mattsson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborgs universitet
Göteborg, Sverige 2025

Slutrapport Grupp 17
Ett modulärt verktyg för att illustrera datastrukturers beteende
Eddie Zell Hedi Kelesh Zakariya Hassan Mandus Högberg Joakim Torstensson Christian Mattsson

Handledare: Jonas Duregård, Institutionen för Data- och Informationsteknik.
Examinator: Patrik Jansson och Arne Linde, Institutionen för Data- och Informationsteknik.
Rättande lärare: Birgit Grohe, Department of Computer Science and Engineering.

Kandidatarbete 2025
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola och Göteborgs Universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Typeset in LaTeX
Göteborg, Sverige 2025

Final report Group 17
A modular tool for illustrating the behavior of data structures
Eddie Zell, Hedi Kelesh, Zakariya Hassan, Mandus Högberg, Joakim Torstensson, Christian Mattson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This project presents a framework for algorithm visualization that emphasizes the separation of execution logic from its visualization. The main research question was how a log-based embedded Domain-Specific Language (eDSL) could be designed to generate a structured log file capable of describing algorithm behavior, without coupling the algorithm code with the visual interface. The project focused on three primary objectives: designing a log format that can represent both individual operations and grouped code blocks, developing a library with minimal impact on the algorithm's structure, and exploring how the log can be used to generate visualizations in a separate frontend. The project has resulted in a TypeScript-based library that includes predefined data structures and variable classes, capable of automatically logging relevant operations. The generated log is in JSON format and contains detailed information such as variable states, scopes and animation types. Although a prototype for visualization was initiated, it was not completed due to time constraints, highlighting an opportunity for future work. Nonetheless, the project demonstrates that a decoupled model for algorithm visualization using a log-based eDSL is entirely feasible.

Keywords: datastructures, algorithms, visualization, embedded language, domain-specific language, programming, computer science

Sammandrag

Detta projekt presenterar ett ramverk för visualisering av algoritmer med fokus på att separera exekveringslogik från dess visualisering. Den övergripande forskningsfrågan var hur ett loggbaserat inbäddat domänspecifikt språk (eDSL) kan utformas för att generera en strukturerad loggfil som beskriver algoritmers beteende, utan att algoritmens kod kopplas samman med det visuella gränssnittet. Projektet fokuserade på tre huvudsakliga mål: att utforma ett loggformat som kan representera både enskilda operationer och grupperade kodblock, att utveckla ett bibliotek med minimal påverkan på algoritmens struktur, samt att undersöka hur loggen kan användas för att generera visualiseringar i en separat frontend. Resultatet är ett TypeScript-baserat bibliotek som innehåller fördefinierade datastrukturer och variabelklasser, vilka automatiskt loggar relevanta operationer. Den genererade loggen är i JSON-format och innehåller detaljerad information såsom variabeltillstånd, scopes och animationstyper. Även om en prototyp för visualisering påbörjades, färdigställdes den inte på grund av tidsbrist, vilket pekar på ett område för framtida arbete. Trots detta visar projektet att en frikopplad modell för algoritmvisualisering med hjälp av ett loggbaserat eDSL är fullt genomförbar.

Nyckelord: datastrukturer, algoritmer, visualisering, inbäddat språk, domänspecifikt språk, programmering, datavetenskap

Förord

Vi vill tacka Jonas Duregård för hans stöd och relevanta råd under arbetet. Jonas hjälp har gjort det här arbetet genomförbart och utan honom hade det här arbetet tagit dubbelt så lång tid.

Eddie Zell, Hedi Kelesh, Zakariya Hassan, Mandus Högberg, Joakim Torstensson, Christian Mattson, Göteborg, juni 2025

Innehållsförteckning

Figurförteckning

Kodförteckning

1

Inledning

Datastrukturer och algoritmer är några av de mest grundläggande koncepten inom datavetenskap. De möjliggör effektiv datahantering och problemlösning, och deras relevans är oberoende av vilket språk eller operativsystem som används. En djup förståelse för dessa koncept är därför avgörande för att kunna utveckla bättre och effektivare algoritmer och program.

För att underlätta denna förståelse skulle därför ett välfungerande verktyg som bryter ner och visualiserar den inre processen hos en datastruktur och dess tillhörande algoritmer vara ett värdefullt stöd. Genom att tydliggöra de underliggande processerna blir det enklare att analysera och kommunicera kring dessa koncept. I en utbildningskontext kan ett sådant verktyg fungera som ett gemensamt och interaktivt hjälpmedel för både lärare och studenter.

Även utanför ramen av utbildning finns det relevanta problem relaterade till felsökning. Utan ett generaliserat visualiseringsverktyg finns det få systematiska metoder för att felsöka en implementerad algoritm eller datastruktur som inte innebär att stega genom kodrader eller att skriva olika tillståndsrelaterade variabler till en terminal - båda vilka ger en begränsad helhetsbild när det kommer till mer komplexa problem. En tydlig och interaktiv visualisering hade därför minskat komplexiteten av felsökning.

Ett av flera verktyg [17] som har utvecklats för detta syfte har tagits fram av Peter Ljunglöf [19], med funktionalitet för att visualisera ett antal olika träd och grafer. Ljunglöfs verktyg, DSVIS, är ett interaktivt visualiseringsverktyg för datastrukturer som används inom kurser i datastrukturer vid Chalmers och Göteborgs universitet. Det största problemet med DSVIS är en brist på separation mellan logik och representation. Detta gör det omständligt att både bygga vidare på och underhålla, då förändringar i datarepresentationen kan medföra omfattande omskrivningar av redan implementerade algoritmer. Vidare bidrar en saknad isolering av logik till en mindre lättläst och begriplig kod.

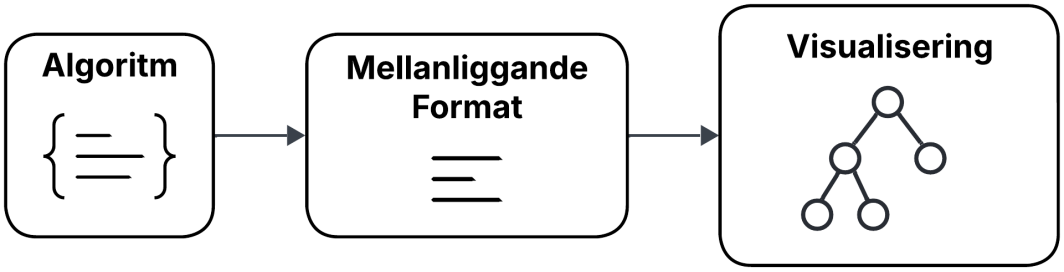
Ett annat problem med verktyget är att möjligheterna att visualisera en viss algoritm bygger på att algoritmen implementerar ett specifikt gränssnitt. I fallet med trädalgoritmer måste exempelvis klassen DSVIS.Engine vara implementerad, eftersom visualiseringen bygger på att algoritmen direkt anropar funktioner i denna klass. Detta skapar en stark koppling mellan algoritmen och visualiseringsgränssnittet, vilket hämmar möjligheterna att återanvända algoritmerna i andra kontexter såsom felsökning eller alternativa visualiseringslösningar.

Ett annat verktyg utvecklat av Juan Lin och Hui Zhang [10] för liknande syfte har en bättre separation mellan visualisering och algoritmens implementering, men har liknande problem relaterat till kopplingen mellan exekvering och visualisering. Istället för att sammanväva algoritmen med visualiseringskommandon, bygger detta verktyg på att exekvera algoritmens relaterade Python-kod [20] och visualisera resultatet från det underliggande tillståndet i minnet. Detta gör att algoritmens implementering inte beror på visualiseringen, medan det samtidigt medför en stark koppling till just Python. Detta försvårar möjligheterna att visualisera algoritmer implementerade i andra språk.

Problemen med dessa lösningar visar tydligt att det finns en möjlighet att utveckla ett verktyg som är mer strukturerat och betydligt lättare att använda och underhålla.

1.1 Syfte

Syftet med detta arbete är att angripa problemen med befintliga kodbaser genom att utveckla ett inbäddat domänspecifikt språk [24] för att uttrycka de olika tillstånd och handlingar en algoritm eller datastruktur genomgår under olika exekveringssekvenser. Målet är att producera en ”mellanliggande representation” av algoritmens exekvering, som sedan kan tolkas av ett visuellt gränssnitt för att skapa en visualisering. Eftersom visualiseringen enbart bygger på den mellanliggande representationen, skapar detta en tydlig separation mellan algoritmen och det visualiserande gränssnittet, vilket gör att både gränssnittet och det valda programmeringsspråket för algoritmen är modulärt utbytbara. Se figur 1.1 för en illustration av det tänkta flödet.



Figur 1.1: En översiktlig illustration av det tänkta flödet för systemet.

1.2 Mål och Frågeställning

Den övergripande frågeställningen för arbetet är: *Hur kan ett loggbaserat eDSL utformas för att möjliggöra visualisering av algoritmers beteende, utan att koppla samman exekveringslogik med presentation?*

För att besvara denna fråga fokuserar projektet på tre delmål:

- Att ta fram ett loggformat som kan beskriva både enskilda körsteg och steg som hör ihop i grupper, som till exempel loopar eller funktioner.

- Att utveckla ett bibliotek som gör det enkelt att skapa sådana loggar för en given algoritm, med minimal påverkan på algoritmens struktur.
- Att undersöka hur denna logg kan användas för att skapa en visuell återgivning av algoritmens flöde i ett separat gränssnitt.

1.3 Avgränsningar

För att säkerställa projektets genomförbarhet inom den givna tidsramen och bibehålla en tydlig koppling till det akademiska sammanhanget, avgränsas arbetet till de datastrukturer och algoritmer som behandlas inom den kurs där visualiseringsverktyget primärt är avsett att användas [3]. Fokus ligger på att utveckla visualiseringar av de datastrukturer och algoritmer som ingår i kursens kursplan och examination.

Specifikt prioriteras visualisering av grundläggande datastrukturer såsom listor, köer, stackar och binära sökträd. När det gäller algoritmer begränsas arbetet till sorteringsalgoritmer såsom insertion sort, merge sort och quicksort, samt sökalgoritmer som linjär och binärsökning. För grafalgoritmer omfattas de som behandlas i kursen [3], exempelvis depth-first search, breadth-first search samt kortaste vägen-algoritmer såsom Dijkstras algoritm.

Utöver omfånget av relevanta algoritmer och datastrukturer har arbetet även begränsats till att enbart hantera implementationer i JavaScript [18] och TypeScript [23].

2

Teori

Detta kapitel behandlar relevant teori som utgör grunden för arbetet. Urvalet förutsätter att läsaren besitter grundläggande kunskaper inom datavetenskap och IT.

2.1 Systemarkitektur inom mjukvaruutveckling

Begreppet systemarkitektur avser en konceptuell modell som beskriver ett systems struktur, beteende och vyer [27]. Inom kontexten av mjukvarusystem talar man ofta om vilka mjukvarukomponenter, såsom klasser, gränssnitt och moduler, som systemet innefattar, samt hur dessa är relaterade och interagerar med varandra och med externa aktörer.

En central aspekt inom mjukvarans systemarkitektur är hur dessa komponenter organiseras och struktureras för att uppnå en modulär design. Målet är att dela upp systemet i 'löst kopplade' och avgränsade moduler som var och en ansvarar för en specifik del av programmets funktionalitet. Med en sådan design skapas tydliga ansvarsområden mellan modulerna, vilket underlättar underhåll, testning och återanvändbarhet markant. Framför allt minskar komplexiteten då en modulär design isolerar förändringar till enskilda komponenter, vilket gör det möjligt att felsöka samt vidareutveckla olika delar av systemet parallellt i olika team.

För att uppnå en modulär och flexibel mjukvaruarkitektur krävs det, utöver organiserad struktur, även att modulerna samverkar på ett effektivt sätt. Detta kan uppnås med hjälp av väletablerade samt beprövade principer och mönster inom datavetenskap.

2.1.1 SOLID-principerna

SOLID är en uppsättning av fem principer inom objektorienterad mjukvarudesign, med syftet att underlätta utveckling genom att göra programkod lättare att underhålla samt att underlätta för vidare expansion. Poängen är att guida utvecklare till bättre arkitektur. De fem principerna är följande:

- *Single-Responsibility Principen* - Innebär att varje modul bör bära ett och endast ett ansvar. När en modul har flera ansvarsområden ökar kopplingen mellan dessa, vilket kan leda till svårigheter vid underhåll och vidareutveckling. Ändringar i ett ansvar kan då oavsiktligt påverka andra delar, vilket ökar risken för fel. [21]

- *Open-Closed Principen* - Innebär att en modul bör vara öppen för utökning men stängd för förändring. Detta innebär att det ska vara möjligt att lägga till ny funktionalitet utan att behöva ändra befintlig kod. Genom att strukturera koden enligt denna princip minskar risken att introducera fel i redan fungerande delar av systemet, samtidigt som systemet förblir flexibelt för framtida krav. [21]
- *Liskov Substitution Principen* - Innebär att objekt av en subtyp ska kunna ersätta objekt av bastypen utan att påverka systemets korrekthet. Med andra ord måste en subtyp bevara kontraktet som definieras av bastypen, vilket innebär att den ska erhålla samma funktionalitet och beteende i alla sammanhang där basklassen används. [21]
- *Interface-Segregation Principen* - Innebär att ingen modul ska tvingas bero på metoder som den inte använder. Det är därmed lämpligare att ha flera små och mer specifika gränssnitt över ett stort heltäckande. Genom att dela upp gränssnitt på detta sätt blir systemet mer modulärt, lättare att underhålla och enklare att testa. [21]
- *Dependency-Inversion Principen* - Innebär att högnivåkomponenter inte ska vara beroende av lågnivåkomponenter, utan att båda ska förhålla sig till gemensamma abstraktioner. Det innebär också att detaljer i implementationen ska bygga på abstraktioner och inte tvärtom. Genom att komponenter förlitar sig på abstraktioner, såsom gränssnitt, istället för konkreta implementationer, uppnås en lösare koppling mellan systemets delar. Detta gör systemet mer flexibelt och både enklare att underhålla och testa. [21]

2.1.2 Designmönster

Inom mjukvarudesign existerar flera återkommande problem som är gemensamma för många olika områden. För att tackla dessa har det tagits fram en mängd generella och återanvändbara lösningar - så kallade *designmönster*. De vanligaste designmönsterna är uppdelade i tre huvudkategorier baserat på vilken typ av problem de är avsedda att lösa:

- *Skapandemönster* - används för att abstrahera ett objekts skapandeprocess och i sin tur säkerställa att objekten skapas på ett kontrollerat och flexibelt sätt.
- *Strukturmönster* - beskriver hur klasser och objekt kan organiseras och kombineras för att bilda större strukturer på ett flexibelt och återanvändbart sätt.
- *Beteendemönster* - fokuserar på kommunikation och ansvarsfördelning mellan objekt för att hantera kontrollflöden och algoritmer.

Nedan följer några vanliga designmönster:

Observatörmönstret
En prenumerationstjänst till ett objekt skapas för att kunna meddela flera andra objekt när förändringar eller uppdateringar sker på det prenumererade objektet [22]. Gällande beroenden i mönstret skapas ett en-till-många beroende mellan objekten.

Mönstrets huvudsakliga användningsområden finns när en uppdatering på ett objekt bygger ett annat objekt och mängden objekt som ska uppdateras är okänd. [7]

Dekoratörmönstret
Ett sätt att lägga till ytterligare funktionalitet för ett objekt dynamiskt genom att kapsla in objektet i ytterligare ett objekt som innehåller den nya funktionaliteten. Dekoratörmönstret ger mer flexibilitet än en subklass då en subklass tillför funktionaliteten statiskt och inte ger användaren någon kontroll över när objektet bör kapslas in. [7]

Composite pattern
Composite-designmönstret används för att hantera hierarkiska datastrukturer där både enskilda och sammansättningar av objekt hanteras som enskilda objekt. Detta mönster appliceras främst på trädliknande datastrukturer och definierar ett gemensamt gränssnittstyp för både löv-noderna och sammansatta noder. Klienter kan då interagera med hela trädet utan att särskilja mellan de olika typerna av noder. [7]

Strategy pattern
Strategy mönstret handlar om att definiera ett flertal algoritmer eller beteenden i separata klasser och göra deras objekt (också kallade strategier) tillgängliga för att användas i en gemensam kontext (klass). Detta främjar flexibiliteten hos programmet då klienten kan byta strategi vid körning utan att det påverkar den omgivande koden i huvudkontexten. [7]

2.1.3 Model-View-Controller (MVC)

MVC står för ”Model-View-Controller” och är ett arkitekturmönster för mjukvarusystem där fokus ligger på att separera logik från dess representation. MVC är inget nytt designmönster; konceptet med att separera olika gränssnitt kan spåras tillbaka till 1980- och 1990-talet [11].

Grundprincipen inom MVC är att en applikation delas upp i tre huvudsakliga komponenter: *Model*, *View* och *Controller*. *Model* innehåller applikationens kärnlogik, det vill säga det som ”modelleras”. Exempelvis kan detta vara spelreglerna i ett schackprogram, en simulering av en motor eller logiken till ett liknande system. [7]

View representerar användargränssnittet och ansvarar för att presentera data från modellen på ett visuellt eller textuellt sätt. En *View* kan till exempel vara en grafisk visualisering av ett schackbräde med alla pjäser, men kan lika gärna utgöras av en textbaserad representation i en terminal. [7]

Controller fungerar som ett gränssnitt mellan användaren och modellen genom att hantera all indata och omvandla den till instruktioner för modellen. *Controllern* kan betraktas som ett mellanliggande lager mellan *View* och *Model* som ansvarar för kommunikationen mellan dessa två. [7]

MVC-mönstret bygger på en strikt separation mellan komponenterna. *Model* bör inte ha något beroende till vare sig *View* eller *Controller*, eftersom sådan koppling skulle bryta mot principen om separation av ansvar. Denna avgränsning möjliggör hög modularitet och underlättar både testning och vidareutveckling av systemet. [7]

En central fördel med detta tillvägagångssätt är att en och samma *Model* kan återanvändas tillsammans med flera olika *Views* och *Controllers*, utan att modellen behöver ändras. Exempelvis kan en modell som representerar ett schackspel både visualiseras i ett grafiskt gränssnitt och via en textbaserad terminalvy, eller styras via olika typer av input. Detta ökar flexibiliteten och gör det enklare att anpassa systemet till olika användarbehov och plattformar. [7]

2.2 Serialisering och deserialisering av data

Vid överföring av programrelaterad data (exempelvis datastrukturer och andra objekt) till en annan domän, exempelvis för långtidsförvaring, krävs att informationen omvandlas till ett format som kan lagras eller överföras. Denna process kallas *serialisering*, och motsvarande process, där serialiserad data återskapas i sitt ursprungliga format, kallas analogt för *deserialisering* [26]. Se kodutdrag 2.1 nedan för ett enklare exempel.

```
class Person {
  private string name;
  private int age;

  public Person(name, age) {
    this.name = name;
    this.age = age;
  }
}

const ada = new Person("Ada", 42);
const jsonString = JSON.stringify(ada);

console.log(jsonString);

// Output:
// {"name": "Ada", "age": 42}
```

Kodutdrag 2.1: Ett enkelt exempel på serialisering av ett JavaScript-objekt till en JSON-sträng.

2.3 JavaScript och TypeScript

JavaScript är ett dynamiskt, interpreterat programspråk som vanligen används för att utveckla interaktiva webbapplikationer. Språket körs i webbläsaren, men används också i servermiljöer via plattformar som Node.js.

Eftersom JavaScript är svagt och dynamiskt typat kan det lätt leda till svårupptäckta fel, då många problem inte upptäcks förrän vid körning av koden, vilket statiska kodanalyser har begränsade möjligheter att identifiera. På grund av dessa svagheter väljer många utvecklare att använda TypeScript, som bygger på JavaScript,

men är ett striktare och statiskt typat språk. TypeScript introducerar typkontroll, gränssnitt och andra språkfunktioner som underlättar skalbarheten och underhåll av större kodbaser. Det kompileras till JavaScript och kan därför användas på samma plattformar. [23]

2.4 JSON

JSON (JavaScript Object Notation) är ett lättviktigt dataformat som används för att strukturera data i en textbaserad form. Formatet är språkoberoende, lätt att läsa för både människa och maskin, och används ofta för dataöverföring mellan system [5]. JSON-syntax bygger på objektsyntaxen i JavaScript bestående av nyckel-värde-par omslutna av klammerparenteser. I dess enklaste form kan en giltig JSON-fil egentligen bara bestå av en siffra eller till och med *NULL*, men den typiska formen av en JSON-fil är just i formen av ett JavaScript-objekt (se kodutdrag 2.2 nedan).

```
{
    "Key1": value,
    "Key2": value,
    "Key3": []
}
```

Kodutdrag 2.2: Exempel på JSON-struktur.

I JSON används ett flertal olika datatyper. Bortsett från de vanligaste datatyperna (String, Number och Boolean) så finns även Null, Array och Objekt. Arrayer i JSON fungerar likt arrayer i många andra språk och noteras med `"[]"`. En viktig distinktion i hanteringen av Null-datatypen är att *Null* och *Undefined* inte är samma sak. I JavaScript så finns datatypen *Undefined* men inte i JSON. JSON är i sin grund ett objekt. När ett nyckel-värde-par skapas i JSON är även det ett objekt [2]. JSON blir därmed ett rekursivt språk och ett exempel på det kan finnas i kodutdrag 2.3 nedan.

```
{
    "Key1": {
        "KeyA": "ValueA",
        "KeyB": {
            "KeyBB": "ValueBB"
        }
    },
    "Key2": "Value",
    "Key3": []
}
```

Kodutdrag 2.3: Exempel på JSON-rekursiv struktur.

2.5 Domänspecifika språk (DSL)

Domänspecifika språk, förkortat DSL från engelskans ”Domain-Specific Language”, är språk som är skräddarsydda för ett visst område eller problem. DSL:er utvecklas på grund av att det kan finnas mer naturliga sätt att uttrycka lösningar för problem än vad som erbjuds med klassiska generella programmeringsspråk [8]. Ett DSL ska innehålla all semantik för en domän och inget mer. Om det innehåller överflödig semantik är språket för generellt och om det innehåller otillräcklig semantik så uppfyller det inte sitt syfte. Exempel på olika DSL:er är HTML, SQL och LaTeX. DSL:er har ett antal fördelar, dessa är följande:

- De är mer precisa
- Kan skrivas snabbare
- Lättare att underhålla
- Lättare att resonera kring

Dessa fördelar lyfts exempelvis fram i [8] [13].

Skillnaden mellan DSL:er och språk kända som General purpose languages (GPL) är att DSL:er är mindre flexibla än GPL:er till förmån för produktivitet och relevans inom en specifik domän. Något som är viktigt att komma ihåg är att DSL:er kan vara mer eller mindre domänspecifika. Språk som HTML anses mindre domänspecifika då det har en väldigt bred domän och innefattar nästan alla typer av webbsidor. Ett mer domänspecifikt hade varit ett språk för implementation av olika kylprogram för ett visst märke av kylskåp[12].

2.5.1 Inbäddade språk

Inbäddade (domänspecifika) språk, även förkortat eDSL eller DSEL från engelskans ”Embedded Domain-Specific Language” respektive ”Domain-Specific Embedded Language”, är språk där istället för att bygga ett språk helt från grunden så använder man sig av designen och funktionaliteten från redan existerande språk. En fördel med detta är överlapp mellan olika domäner eftersom språken har mycket gemensamt. Ett mycket kort exempel på ett eDSL i Python hade kunnat vara:

```
class Add:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def eval(self):
        return self.a + self.b

class Sub:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def eval(self):
        return self.a - self.b
```

```
class Mul:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def eval(self):
        return self.a * self.b

expr = Mul(Add(2, 3), Sub(4, 1))
print(expr.eval())  # Outputs: 15
```

Istället för att skriva: `result = (2 + 3) * (4 - 1)` så kan det nu skrivas som `expr = Mul(Add(2, 3), Sub(4, 1))`.

3

Metod

3.1 Övergripande systemarkitektur

I linje med principerna för MVC identifierades till en början två huvudsakliga komponenter i systemarkitekturen - algoritmen (modell) och visualiseringen (vy). Målet var att i största mån hålla algoritmens exekvering och det visuella gränssnittet oberoende av varandra. För att uppnå detta definierades ett mellanliggande JSON-format som skulle fungera som ett gränssnitt mellan en algoritm och dess representation. Detta format beskriver de olika tillstånd som en algoritm och dess datastrukturer genomgår under exekveringen, samt de operationer som utförs mellan tillstånden. Visualiseringen är således inte beroende av algoritmens implementation, utan enbart av dess serialiserade representation.

Denna design ledde till tre huvudsakliga spår för arbetet:

- Utveckling av ett JSON-baserat DSL för att representera exekveringssteg.
- Utveckling av ett TypeScript-bibliotek för att logga exekveringsdata från algoritmer.
- En webbaserad visualiseringsmiljö som tolkar loggen och genererar animationer.

3.2 Utformning av JSON-baserat DSL

För att möjliggöra en tydlig uppdelning mellan exekveringslogik och presentation togs ett domänspecifikt språk (DSL) fram i JSON-format. DSL:et beskriver algoritmers beteende i form av strukturerade loggar, vilket gör det möjligt att visualisera körningar oberoende av programmeringsspråk eller implementation.

Målet med DSL:et var att göra det möjligt att i detalj beskriva vad som händer under en algoritms körning, så att visualiseringen kan styras enbart utifrån loggdata. Visualiseringen bygger därmed inte på tolkning av kod, utan tolkning av strukturerad JSON-data. För att detta skulle vara möjligt behövde DSL:et uppfylla följande krav:

- Samtliga relevanta tillstånd i algoritmen (datastrukturer, variabler, etc.) måste kunna representeras vid varje exekveringssteg.
- Förändringar mellan tillstånd måste vara uttryckta som separata operationer (animationssteg) för att möjliggöra animering.

- Steg måste vara reversibla för att stödja navigering både framåt och bakåt i visualiseringen.
- Funktionella strukturer såsom funktionsanrop och block (scopes) måste kunna representeras för att bevara kontext.

Utifrån dessa krav utformades ett format där varje algoritmkörning primärt består av en lista av steg, kallad *steps*, som i sin tur innehåller information om tillstånd, operationer och kontext.

3.2.1 Struktur för exekveringssteg

Grunden i DSL:ets struktur är en lista av exekveringssteg, representerad av fältet *steps*. Varje steg i loggen beskriver ett specifikt exekveringsmoment i algoritmens körning och innehåller tillståndsinformation för relevanta datastrukturer och variabler. Genom att dela upp exekveringen i diskreta, isolerade steg kan man skapa en detaljerad och reproducerbar logg som gör det möjligt att visualisera algoritmens beteende i detalj.

Varje steg i listan består av en av tre följande typer:

- **Tillståndsdump:** En beskrivning av alla relevanta datastrukturer och variabler med tillhörande tillstånd vid det aktuella steget.
- **Animationssteg:** En beskrivning av förändringen som skedde mellan två intilliggande steg. Till exempel en jämförelse mellan två värden eller ett platsbyte i en array.
- **Scope:** Ett scope representerar ett kodblock som är tänkt att innehålla en avskild kontext, såsom ett funktionsanrop, och kan i sin tur innehålla egna *steps*.

Loggen är alltså inte enbart en sekvens av tillstånd, utan även en detaljerad beskrivning av de transformationer som sker mellan tillstånden. Detta gör det möjligt att animera förändringar istället för att bara visa resultat.

Denna struktur möjliggör dessutom att loggen kan spelas upp både i framåt- och bakåtriktning. För detta krävs att varje steg innehåller tillräcklig information för att kunna återskapa både sitt eget tillstånd och den förändring som lett fram till det.

3.2.2 Animationssteg

För att visualiseringen ska kunna ge en tydlig bild av de operationer som sker mellan dem, infördes så kallade *AnimationStep* (animationssteg). Ett animationssteg beskriver explicit vilken typ av förändring som skett mellan två intilliggande tillstånd. Exempelvis kan detta vara en jämförelse av två element eller ett platsbyte i en array.

Varje animationssteg består av:

- En **typ**, såsom ”swap”, ”push”, ”pop”, ”enqueue” eller ”dequeue”.

- En eller flera **motiv**, som beskriver vilka element som berörs av operationen, till exempel index eller ID:n av andra datastrukturer.
- **data**, som beskriver eventuell information som är relevant till operationen som utförs och/eller de objekt som ingår i animationen.

3.2.3 Markeringar och variabelpekare

En central del i förståelsen av algoritmers exekvering är att kunna följa vilka element som är aktiva i ett givet exekveringssteg. Därför infördes ett frivilligt fält i varje scope-steg kallat *highlight*, som gör det möjligt att markera specifika datastrukturer och element.

Fältet *highlight* är en lista bestående av två olika typer:

- **Datastrukturens ID** - för att identifiera vilken datastruktur som är markerad.
- En tvåtupel bestående av:
    1. **Datastrukturens ID** - för att identifiera vilken datastruktur som är markerad.
    2. **Index** - vilket element i strukturen som ska markeras.

Detta gör det möjligt att i ett enda steg markera flera olika element i en eller flera datastrukturer samtidigt och även hålla dessa markerade över flera substeg. Markeringarna kan användas av visualiseringen för att rita pilar vid jämförelse av två element eller färgmarkera viktiga värden i en sökträdstruktur.

Utöver markeringar stödjer loggen även en representation av *variabler som pekar på positioner i datastrukturer*. Detta är särskilt användbart i algoritmer som använder sig av rörliga index, såsom sorteringsalgoritmer (t.ex. *left* och *right* i en typisk implementation av Quicksort). Genom att logga sådana variabler som referenser till specifika index i en struktur kan visualiseringen rita ut pekare som rör sig över olika steg, vilket ger en tydligare inblick i hur algoritmen arbetar.

Det är dock viktigt att dessa pekarvariabler även kan användas som vanliga variabler - exempelvis för att avgöra när en iteration ska avslutas.

3.2.4 Scopes och funktionsanrop

För att modellen ska kunna representera en realistisk exekvering av algoritmer med separata/individuella kontexter för blockstruktur, funktionsanrop och rekursion, behöver DSL:et stödja en form av hierarki i exekveringsflödet. Detta genomfördes genom stegtypen *scopes*, vilket motsvarar avgränsade kodblock, exempelvis loopar, villkor eller funktioner. Varje scope kan innehålla egna exekveringssteg, egna variabler och tillstånd, samt egna animationssteg.

Syftet med scopes är att isolera förändringar till den del av algoritmen där de uppstår, för att underlätta möjligheterna till kontextualiserad visualisering, och samtidigt återspegla den logiska blockstruktur som finns i källkoden. Detta ger en direkt

koppling mellan kod och visualisering - användaren kan exempelvis se att en viss operation sker *inuti* en while-loop eller *under* ett funktionsanrop.

Funktionsanrop
En särskilt utmaning uppstår vid visualisering av funktionsanrop, särskilt i algoritmer som använder rekursion. Vid varje anrop bör en ny, isolerad miljö skapas där endast de relevanta argumenten och lokala variabler är synliga. Detta säkerställer att loggen återspeglar det stack-beteende som funktionella exekveringsmodeller bygger på.

För att uppnå detta infördes följande konventioner:

- Varje funktionsanrop genererar ett nytt scope i loggen. Detta scope innehåller de initiala argumenten till funktionen som variabler, givet att dessa är loggade objekt, samt ett eget steps-fält för att beskriva exekveringen inom anropet.
- Lokala variabler som deklareras inom ett funktionsanrop finns endast tillgängliga inom det scope där de skapats. Efter att funktionen avslutats är dessa variabler inte längre åtkomliga i efterföljande steg.
- Även om det lokala tillståndet isoleras, kvarstår effekter på delade datastrukturer efter att funktionen exekverats. Om en funktion exempelvis modifierar en array som skickats in som argument, speglas denna ändring i efterföljande steg utanför scopet.

Genom att modellerna stöder denna typ av anropsstruktur blir det möjligt att visualisera även komplexa algoritmer, exempelvis Quicksort eller Depth-First Search, utan att tappa separationen mellan kontexter i olika nivåer av exekveringen.

Nestlade scopes
Scopes kan vara rekursivt nestlade, vilket innebär att ett scope kan innehålla ytterligare scopes, t.ex. om en funktion anropar sig själv, eller om en loop innehåller ett villkorsblock. Detta ger loggen en trädliknande struktur som speglar det verkliga kontrollflödet hos algoritmen. Visualiseringen kan därmed anpassas för att ge en tydlig separation av olika kontexter i olika rekursionsnivåer.

3.3 Typescript bibliotek

För att det mellanliggande DSL-formatet skulle kunna användas praktiskt krävdes det ett verktyg som kunde generera korrekt strukturerade loggar från givna kodstycken. På grund av detta utvecklades ett TypeScript-bibliotek som går att använda för att generera en JSON-logg av algoritmer i enlighet med den specifikation som beskrivits tidigare.

Utvecklingen av biblioteket började med att skapa en central logger klass som skulle sköta loggningen av algoritmerna, med målet att ha så lite inverkan på algoritmens kod som möjligt. Därför gjordes det en tidig ansats att använda Proxy-objekt [15] i TypeScript. Tanken bakom detta designval var att man hade kunnat låta datastrukturer och variabler omges av en proxy som i sin tur applicerar alla operationer på det underliggande objektet, samtidigt som alla förändringar relevanta till algoritmen loggades av ett logger-objekt. På så sätt hade koden till algoritmen haft

minimal påverkan och all spårning och dokumentation hade skett semi-automatiskt via proxyn.

Men för arbetets syfte visade sig detta vara begränsande. Problemet vi stötte på var att objekt inuti proxys i TypeScript inte fungerar på samma sätt som vanliga variabler vid tilldelning. Det går inte att ersätta hela det objekt som en proxy representerar eftersom proxyn i sig är ett objekt, och proxyns interna referens till det underliggande värdet inte automatiskt uppdateras vid tilldelning. För att byta ut det underliggande värdet krävs istället att man refererar till proxyns interna fält och tilldela det ett annat värde.

Eftersom begränsningen försvårade flexibel loggning, valde gruppen att ersätta proxy-lösningen med ett mer strukturerat tillvägagångssätt. Detta tog formen av ett flertal TypeScript implementationer av diverse datastrukturer som har loggnings-funktionalitet från den centrala Logger-klassen, för att spåra förändringar i deras tillstånd. Genom att låta loggningen ske på så sätt blev det enklare att anpassa loggningen till de specifika behov olika algoritmer och datastrukturer har.

3.3.1 LoggedArray

LoggedArray blev representationen av arrayer med logg funktionalitet. I klassen finns funktioner som utför operationer på den underliggande arrayen, samtidigt som det dokumenteras som ett steg i den gemensamma loggen. De elementära operationerna som implementerats är set() och get() som ger tillgång till ett specifikt index, för att hämta eller skriva in värden. Eftersom platsbyten i arrayer är en operation som utförs i många algoritmer finns det som en egen funktion, swap(), och dokumenteras som ett animationssteg för att möjliggöra dess visualisering.

3.3.2 LoggedVariable

Det implementerades även en representation av variabler med loggfunktionalitet i klassen Loggedvariable. Denna klass används för att representera variabler som behöver spåras genom exekveringen, exempelvis pekare eller temporära värden. I en visualisering var det tänkt att de visades som exempelvis färgmarkeringar eller pilar. De operationerna som finns för en LoggedVariable är set(), get() samt modify(). Modify() fungerar på så vis att det applicerar en lambda-funktion [14] på en LoggedVariable som till exempel a.modify(x => x+1).

3.3.3 LoggedBST

LoggedBST var avsedd att representera ett binärt sökträd med loggfunktionalitet. Klassen hanterar det strukturella tillståndet av trädet, alltså noderna och dess kanter, samt förändringar i dem. Dessutom innehåller klassen funktioner för att utföra en viss operation på trädet vilket också loggas vid exekveringen. De funktionerna som finns i klassen är insert(), som sköter insättning av en nod i trädet i enlighet med reglerna för insättning hos binära sökträd och delete() som tar bort en nod på samma sätt.

3.3.4 LoggedGraph

Slutligen togs även LoggedGraph fram för att representera grafer med loggfunktionalitet. Denna implementation byggdes för att stödja just riktade grafer med dess noder och kanter samt vikter. Funktionerna som blivit implementerade är addNode(), som lägger till en nod till grafen, addEdge() som lägger till en kant, getNodes() som hämtar alla noder, getEdgesFromNode() som returnerar alla kanter kopplade till en nod och getAllEdges() som returnerar samtliga kanter i grafen. De operationerna som är loggade är de som utför en modifiering på grafen, alltså addEdge och addNode.

3.4 Visualisering

Under arbetets gång har det sen tidigt skede diskuterats hur man hade kunnat visualisera en datastruktur och de operationerna man utfört på den från JSON DSL:et. Visionen för hur ett sådant verktyg skulle se ut var överenskommen, och tanken blev då att en frontendapplikation skulle utvecklas i takt med DSL:et. Applikationen skulle ha ett användargränssnitt likt Peter Ljunglöfs verktyg[19], där en användare skulle kunna välja en algoritm samt föra in data för att sedan klicka på en play-knapp för att spela animationer. Användaren skulle även kunna pausa samt klicka sig fram och tillbaka genom stegen i en algoritms exekvering.

Men denna vision kunde endast delvis implementeras. Arbetet med backenden och dess logikstruktur samt DSL:et prioriterades och frontendens funktionalitet kvarstod vid en grundläggande nivå. Det som visualiseras är tillstånden hos array datastrukturer samt pekare variabler. Frontenden som utvecklats hanterar alltså endast stegen i DSL:et som representerar nuvarande tillstånd, men inte vilka operationer som skett. Det sker även inga animationer för de tänkta operationerna som exempelvis swap där två element skulle animeras då de byter plats.

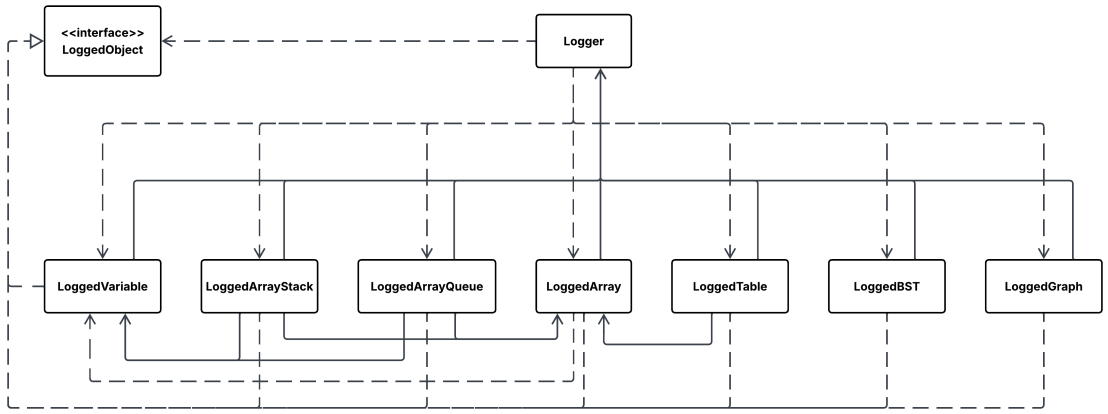
4

Resultat

I detta kapitlet kommer det slutgiltiga verktyget presenteras med hänsyn till dess olika delar.

4.1 TypeScript bibliotek

Den fullständiga implementationen av det utvecklade TypeScript-biblioteket, inklusive dess komponenter för logghantering och strukturering av exekveringsdata finns tillgänglig i projektets kodbas [1].



Figur 4.1: Ett översiktligt UML-diagram av biblioteket.

4.1.1 Logger

Logger klassen innehåller den centrala logiken för projektet. I Logger.ts [1] finns logiken för att skapa ett logger objekt och med objektet kan i sin tur en logg för den datastruktur som är relevant skapas. Se kodutdrag 4.1.

```
1 const logger = new Logger();
2 //initial state of the log: { "steps": [] }
3
4 const array = logger.createArray([1,2,3]);
```

Kodutdrag 4.1: Exempel på initialisering och användning av Logger klassen.

Varje datastruktur och variabel som skapas från ett logger-objekt loggas som ett nytt steg. I detta steg uppdateras listan av existerande variabler och objekt med det nya elementet tillagt. Varje objekt i loggen initialiseras med ett unikt id för att kunna refereras till. Se kodutdrag 4.2. När ett funktionsanrop kan förändra tillståndet i ett loggat objekt bildas också ett nytt steg i loggen. Det steget kommer då vara identiskt till det föregående steget, där endast det loggade objektet som påverkats blir uppdaterat.

```
{
  "steps": [
    {
      "type": "StateDump",
      "state": [
        {
          "description": "array",
          "type": 0,
          "value": [1,2,3],
          "id": 0
        }
      ]
    }
  ]
}
```

Kodutdrag 4.2: Exempel på loggning av ett deklarerat objekt.

Logger klassen ger även möjligheten att använda scopes. Ett scope skapas med hjälp av funktionen scope(), tillsammans med en lambda funktion som parameter, där den koden som exekveras placeras inuti funktionskroppen. Se kodutdrag 4.3 för exempel på hur scope används.

```
const logger = new Logger();
let x = logger.createVar(3);
while (x.get() > 0) {
  logger.scope(() => {
    logger.createVar(5);
  });
}
```

Kodutdrag 4.3: Exempel på användning av scoping.

Precis som med scope används highlight() för att markera variabler och element i datastrukturer under ett godtyckligt långt kodstycke. Koden placeras, likt scope()-funktionen, i en lambda funktion, och på så sätt markeras specifika element under exekveringen av ett kodblock vilket återspeglas i loggen.

```
const array = logger.createArray([1,2,3,4,5];
logger.highlight([[array, 4]],() => {
  ...
});

```

```
/* resulting log excerpt
...
{
   "type": "Scope",
   "subSteps": [...],
   "highlighted": [[0,4]]
},
...
*/
```

Kodutdrag 4.4: Exempel på användning av highlight.

För att logga funktionsanrop använder man functionCall(). Vid rekursiva funktioner skapar varje anrop automatiskt ett nytt scope, vilket innebär att variabler och datastrukturer som deklareras inuti funktionen hålls isolerade från tidigare och efterföljande anrop. Detta förhindrar oönskad delning av tillstånd och möjliggör en korrekt återgivning av algoritmens exekveringsflöde. Resultatet blir en trädstruktur av scopes i loggen, där varje gren motsvarar ett specifikt funktionsanrop.

```
function mergeSort<V>(array: LoggedArray<V>): LoggedArray<V>
   {
  if (array.size() <= 1) return array;

  const [left, right] = array.split(Math.floor((array.size())
     /2));
  const sortedLeft  = Logger.functionCall(mergeSort, left);
  const sortedRight = Logger.functionCall(mergeSort, right);

  return Logger.functionCall(merge, sortedLeft, sortedRight);
}
```

Kodutdrag 4.5: Exempel på en rekursiv funktion.

När man vill generera en JSON-fil av loggen anropar man write() med en sträng som innehåller namnet man vill ge loggen. En fil skapas sedan med samma namn som den angivna strängen. Om det redan finns en fil med samma namn skrivs den över av detta anrop. Denna funktion anropar man på det logger objekt man utfört loggningen med. Fortsatta loggningar som utförs med eller via samma logger objekt utökar den tidigare loggen med nya steg. För att börja på nytt initialiserar man istället ett nytt logger objekt.

```
...
logger.write("log1.json");
// A file with the name 'log1.json' gets generated.
...
```

Kodutdrag 4.6: Generering av en logg.

4.1.2 Arrayer

För att logga ett platsbyte mellan två element i en LoggedArray används swap() funktionen som är definierad som följande i kodutdrag 4.7.

```
public swap
(
  i: number | LoggedIndex<number>,
  j: number | LoggedIndex<number>
): void
{
  if (i instanceof LoggedIndex) i = i.get();
  if (j instanceof LoggedIndex) j = j.get();
  [this.array[i], this.array[j]] = [this.array[j], this.array
     [i]];

  //update log
  const animationStep = {type: "swap", subjects: [this.id],
     data: [i,j]};
  this.logger.logAnimation(animationStep);
  this.logger.logChange(this.id, [...this.array]);
}
```

Kodutdrag 4.7: Definition av swap().

Swap lägger till ett animationStep i loggen för att tydliggöra för frontenden att det bör ske en animation som visar förflyttningen av elementen.

4.1.3 Stackar och köer

Stackar och köer använder båda LoggedArray internt för att representera deras tillstånd. I kombination med LoggedVariable fulländas deras unika beteende. För att illustrera detta finns implementationen av enque() för köer i kodutdrag 4.8

```
public enqueue(item: T): void {
  if (this.items == this.capacity) {
    throw new Error("Queue is full");
  }

  const animationStep = {
    type: 'enqueue',
    subjects: [this.queue.id],
    data: [item, this.head.get()]
  };

  this.getLogger().logAnimation(animationStep);
  this.queue.set(this.head, item);
  this.head.set((this.head.get() + 1) % this.capacity);
  this.items++;
}
```

Kodutdrag 4.8: Definition av enqueue().

4.1.4 Grafer

I LoggedGraph finns funktionerna för att manipulera en graf implementerade där förändringarna till grafen loggas. Dessa funktioner är addNode() och addEdge() som båda kommer att dokumenteras i loggen som en förändring likt LoggedArray-klassen.

```
1 const graph = logger.createGraph<string,number>();
2 graph.addNodes(["A","B","C"]);
3 graph.addEdge("A","B",2);
```

Kodutdrag 4.9: Exempel på initialisering och användning av LoggedGraph.

4.2 Exempelanvändning på olika algoritmer

4.2.1 Quicksort

Quicksort-algoritmen utför operationer som behöver dokumenteras i loggern, som till exempel swap(). Eftersom swap() funktionen dokumenterar operationen automatiskt, som tidigare visats i kodutdrag 4.7, räcker det med att kalla på swap(). I kodutdraget 4.10 nedan kan man se hur swap() funktionen används i QuickSort-algoritmen.

```
1  function quickSort<T>(array: LoggedArray<T>) {
2    quickSortHelper(array, 0, array.size()-1);
3  }
4
5  function quickSortHelper<T>(array: LoggedArray<T>, low:
       number, high: number) {
6    if (low >= high) return;
7    let left  = array.createIndex(low, "left");
8    let right = array.createIndex(high-1, "right");
9    const pivotIndex = array.createIndex(medianOfThree(array,
       low, high), "pivotIndex");
10
11   array.swap(pivotIndex, high);
12
13   while(left.get() <= right.get()) {
14     //find left number
15     while ((left.get() <= right.get()) && (array.get(left) <
         array.get(high))) {
16       left.modify(x => x+1);
17     }
18     //find right number
19     while ((right.get() >= left.get()) && (array.get(right) >
         array.get(high))) {
```

```
20        right.modify(x => x-1);
21      }
22
23      if (left.get() < right.get()) {
24        array.swap(left, right);
25        left.modify(x => x+1);
26        right.modify(x => x-1);
27      }
28    }
29    //swap pivot back to middle
30    array.swap(left, high);
31
32    Logger.functionCall(quickSortHelper, array, low, left.get()
        -1);
33    Logger.functionCall(quickSortHelper, array, left.get()+1,
        high);
34  }
35
36  left.set(left.get()+1);
```

Kodutdrag 4.10: Exempelanvändning på QuickSort

4.2.2 Breadth-First Search för grafer

Den främsta användningen av loggfunktionaliteten hos grafer är att markera de noder och kanter som är aktuella under traverseringen. Nedan finns ett kodutdrag 4.11 för dokumentering av Breadth-First Search-algoritmen på en BST.

```
1  function graphBFS<V,E>(start: V, graph: LoggedGraph<V,E>): V
       [] {
2    const logger = graph.getLogger();
3    const bfsPath: V[] = [];
4    const visitedTable = logger.createTable<V,boolean>();
5    const queue = logger.createQueue<V>(graph.getAllEdges().
       length);
6
7    graph.getNodes().forEach(node => visitedTable.set(node,
       false));
8    visitedTable.set(start, true);
9    queue.enqueue(start);
10
11   while (queue.size()) {
12     const currentNode = queue.dequeue();
13     bfsPath.push(currentNode);
14
15     for (const [edge, pointer] of graph.getEdgesFromNode(
         currentNode)) {
16       logger.highlight([[graph, pointer]], () => {
17       if (!visitedTable.get(edge.dst)) {
18         queue.enqueue(edge.dst);
```

```
19          visitedTable.set(edge.dst, true);
20        }
21        });
22      }
23    }
24
25    return bfsPath;
26  }
```

Kodutdrag 4.11: Exempelanvändning på Breadth-First Search

För att illustrera hur en markering (highlight) av en kant hos en graf ser ut, finns ett exempelutdrag från JSON loggen som genereras. Se kodutdrag 4.12.

```
1  ...
2  {
3    "type": "Scope",
4    "subSteps": [
5      {"type": "enqueue", "subjects": [2], "data": [1,1]},
6      {
7        "type": "StateDump",
8        "state": [
9          {
10           "type": 0,
11           "value": {
12             "vertices": ["A","B","C","D","E"],
13             "edges": [{"src": "A","dst": "B"}, ...],
14             "weights": [...]
15           },
16           "id": 0
17         },
18         ...
19       ]
20     },
21     ...
22   ],
23   "highlighted": [[0,0]]
24 },
25 ...
```

Kodutdrag 4.12: Utdrag av JSON logg för breadth first search

5

Diskussion

Det här kapitlet diskuterar problem som gruppen har stött på under projektet, poängen med projektet och generella tankar om projektets genomförande och resultat.

5.1 Reflektion på resulterande verktyg

Under arbetets gång behövde ramverkets funktionalitet och användarvänlighet övervägas. Ett viktigt mål för ramverket var att loggningen av en algoritm inte skulle präglas av mer kod än nödvändigt. Förhoppningen var att den loggade koden skulle efterlikna den ursprungliga algoritmkoden så mycket att man hade kunnat identifiera algoritmen den är byggd på.

De åtgärder som vidtagits kretsade kring att främst minimera den mentala belastningen för en användare av biblioteket och på så sätt minska felaktiga loggningar. Biblioteket har därför utformats med fördefinierade datastrukturer som loggar sig själva. För projektets primära ändamål är det resulterande biblioteket en snabb lösning som är effektiv för att logga och visualisera grundläggande datastrukturer och algoritmer. Men i längden är utökningsbarheten mycket ohållbar och begränsad.

5.1.1 Biblioteket

Några nackdelar med det presenterade biblioteket är att det inte kan användas på färdigskrivna algoritmer och datastrukturer. Istället är man tvungen att modifiera existerande kod för att logga och visualisera det vilket bryter mot Open-Closed-principen. Bibliotekets fördefinierade klasser implementerar heller inte alla funktioner som deras ursprungliga struktur består av vilket tvingar algoritmer som använder sig av dessa inbyggda funktioner att skrivas om.

Eftersom loggade variabler är objekt medföljer dessutom syntaktiska kostnader i koden. Enkla och korta operationer som att inkrementera värden blir istället långa funktionsanrop. Åtkomst och användning av loggade variabler kräver också funktionsanrop som kan skapa distraktioner från den aktuella koden, särskilt om de används i långa uttryck. Det påverkar även hur indexering fungerar i LoggedArray, särskilt i samband med loggade indexvariabler. Vid indexering av LoggedArray angavs loggade indexvariabler först med i.get() som inparameter. Detta ansågs som ännu ett omständligt sätt att skriva koden, och då skrevs get()-och set()-funktionerna hos LoggedArray om för att kunna ta emot variabeln i sin helhet istället. Den resulterande koden blev då array.get(i).

Detta skapade dock en annan teknisk komplikation för när man hade velat ge en inkrementering av variabeln som inparameter. Eftersom loggade variabler är objekt kan man inte inkrementera dem som vanligt, alltså är man ändå tvungen att använda get() även där, vilket blir array.get(i.get() + 1).

De loggade variablerna följer inte heller alla språkreglerna som TypeScript och JavaScript bygger på. Primitiva variabler som deklareras med nyckelordet 'const är omöjliga att uppdatera, men om en loggad variabel deklareras med 'const' är det alltid möjligt att uppdatera den, även om variabeln innehåller ett primitivt värde.

5.1.2 Prioritering

Bland de datastrukturer som implementerats saknas en viss funktionalitet, och några har inte implementerats alls. Träd har delvis implementerats i form av en loggad BST. LoggedBST-klassen kan användas för att generera en logg över operationer som utförts på ett träd, men loggen som genereras är inte i nuläget helt felfri på grund av tidsbrist samt att det prioriterats lägre än de andra datastrukturerna.

Vidare finns det datastrukturer som inte blivit implementerade alls, däribland linked lists, hash-tabeller samt trädstrukturer som ingår i kursen men inte implementerades, såsom AVL-träd, prioritetsköer och liknande. Här var det även en fråga om tid och prioritering. På grund av deras mer komplicerade struktur ansågs det att de grundläggande datastrukturerna, såsom arrayer och grafer, var viktigare att implementera.

Vid loggning av algoritmer fungerar loggen för de flesta algoritmerna som använder sig av LoggedArray, men loggen är utformad just för sekventiell exekvering. Detta innebär att bara ett steg kan representeras åt gången. Därför saknas för tillfället stöd för att logga algoritmer med asynkron logik eller parallella processer. Detta hade varit användbart för att möjliggöra visualisering av fler typer av moderna algoritmer, som exempelvis parallell sökning eller asynkrona sorteringsmetoder såsom Samplesort [25].

Ett område som har haft ganska låg prioritet under arbetet är det faktiska användarperspektivet. Det har inte genomförts några användartester av visualiseringarna, och det har inte heller skett någon djupare utvärdering av pedagogisk effekt eller visuell presentation. Anledningen till detta är att fokus huvudsakligen legat på utvecklingen av biblioteket och på sätt prioriterats högre.

5.2 Jämförelse med andra verktyg

Det finns ett flertal befintliga verktyg som i viss utsträckning kan jämföras med det system som har utvecklats. Två exempel på sådana verktyg är Immer [16] i JavaScript samt deepdiff [6] i Python. Dessa verktyg är användbara för att identifiera och logga förändringar i datastrukturer, exempelvis vilka värden som har uppdaterats, lagts till eller tagits bort.

Det är dock viktigt att poängtera att dessa lösningar enbart fokuserar på att registrera vad som har förändrats, inte varför förändringen har skett. De saknar alltså

stöd för att spara metadata kring ändringsorsaker, kontextuell information eller bakomliggande intentioner. Det är även begränsat stöd för att hantera temporära variabler, inkomplett data eller mer komplexa datastrukturer där tillfälliga tillstånd kan vara svåra att tolka utan ytterligare analys eller antaganden.

Ett av målen med arbetet var att språket skulle kunna användas i utbildningssyfte. För tillfället finns det tre huvudsakliga verktyg som används för visualisering i kursen om datastrukturer och algoritmer på Chalmers: Galles visualiseringar av algoritmer [4], Gnarley Trees av Kubo Kovac [9], samt VisuAlgo [28]. Likt Ljunglöfs verktyg som nämns i kapitel 1 så är dessa verktyg renodlade visualiseringsprogram och skiljer sig därmed från detta arbete. Samtliga av verktygen som för tillfället används i kursen täcker in ett större antal algoritmer och datastrukturer än vad det DSL som utvecklats gör. Poängen med arbetet var dock inte själva visualiseringen utan det domänspecifika språket. Att kunna skriva en algoritm och sen få en genererad logg är den stora skillnaden. Loggen som genereras kan sedan användas för att skapa liknande verktyg som de som redan finns om så önskas.

5.3 Användningsområde

Det huvudsakliga användningsområdet för det utvecklade verktyget är inom utbildningssammanhang, där det syftar till att stödja inlärningen av datastrukturer och algoritmer genom att tillhandahålla en tydlig och strukturerad visualisering av exekveringsförlopp. Genom att presentera algoritmers och datastrukturers tillståndsförändringar som en sekventiell och trädliknande logg möjliggörs en stegvis och hierarkisk förståelse för hur olika operationer samverkar och påverkar det övergripande flödet. Detta ger studenter en konkret och visuell modell att förhålla sig till, vilket i sin tur kan främja djupare förståelse och analytisk förmåga. För lärare och handledare innebär det en möjlighet att på ett mer systematiskt sätt illustrera abstrakta koncept under föreläsningar, seminarier och laborationer.

Utöver den pedagogiska nyttan kan verktyget även användas inom programvaruutveckling för att analysera och förstå exekveringssekvenser i algoritmer och datastrukturer. Genom att generera en domänspecifik logg som inte är beroende av faktisk körning eller funktionsanrop kan utvecklare isolera och visualisera specifika tillstånd och operationer, vilket underlättar felsökning och förbättrar möjligheten att upptäcka logiska fel. Detta kan leda till mer robusta implementationer och minska den tid som krävs för testning och validering av komplexa funktioner.

På längre sikt kan detta verktyg användas inom många olika områden av mjukvaruutveckling, där det kan minska testtiden och hjälpa till att identifiera problem i kodens flöde och prestanda. I en företagsmiljö kan detta verktyg användas för att visualisera komplexa algoritmer i system som hanterar stora datamängder, vilket kan hjälpa utvecklare att upptäcka ineffektiva delar av kod eller algoritmer som kan orsaka flaskhalsar.

Förutom de direkta användningsområdena inom utbildning och utveckling ger verktygets flexibilitet och anpassningsbarhet även möjlighet att bidra till forsknings- och undervisningsverktyg som kan hjälpa användare att simulera och visualisera

mer komplexa algoritmer och datastrukturer. Detta skulle ge en djupare insikt i hur olika algoritmer fungerar, och förbättra analysen av deras beteende i praktiska applikationer.

5.4 Samhälleliga och etiska aspekter

Detta arbete har fokuserat på den tekniska delen av visualiseringen och något mindre på användaren. En aspekt som en användare lätt kan glömma när den presenteras med ett verktyg är att verktyget inte är bättre än användaren. Det går alltså att skriva logiskt inkorrekta algoritmer utan att det inbäddade språket kommer att klaga. Det går därmed att lära sig fel vilket blir motsatsen mot vad syftet med verktyget är.

Det här är ett arbete utan någon större miljöpåverkan. Den eventuella miljöpåverkan detta projektets påverkan i form av datoranvändning är försumbar. Ett inbäddat språk kommer inte ändra energiförbrukningen hos användaren, inte heller kommer det att i sig självt förbruka någon energi.

Arbetet har inte någon större ekonomisk påverkan då det inte är någon kommersiell tjänst som erbjuds, inte heller har det resulterat i någon ekonomisk vinning eller förlust för oss.

5.5 Framtida arbete

Arbetet har lagt en stadig grund för syftet att logga och visualisera algoritmer, men det finns ett stort antal potentiella förbättringar som kan läggas till och utvecklas.

- **Stöd för asynkrona och parallella algoritmer.** Ett steg i vidareutveckling vore att utöka loggern för att kunna hantera flera samtidiga exekveringsflöden. I praktiken hade detta kunnat ta formen av parallella scopes, eller en implementation av tidslinjer hos biblioteket samt JSON DSL:et för att beskriva samtidighet.
- **Fler datastrukturer.** Flera datastrukturer som linked lists, AVL-träd och prioritetsköer saknas och bör implementeras för att öka täckningen.
- **Förbättrad loggning av variabler och datastrukturer.** En förbättring i hur variablerna och datastrukturerna loggas hade kunnat minimera de syntaktiska kostnaderna. Detta genom att implementera proxys på ett sätt som löser det tidigare nämnda problemet, eller genom annan 'operator overloading' i språk som stödjer det. Med hjälp av en sån lösning hade biblioteket kunnat användas utan att algoritmerna behövts skrivas om i den utsträckningen de gör i det nuvarande verktyget.
- **Fler loggtyper.** Utöver de nuvarande typerna som highlight och animationsteps, kan loggen utökas till att stödja loggade kommentarer för att förklara stegen som sker, samt breakpoint-markeringar för att exempelvis kunna användas i felsökning och felhanteringssyfte.

- **Striktare struktur i JSON DSL:et.** JSON DSL:et har i nuläget mycket generella stegtyp, vilket gör det flexibelt men svårare att användas av en frontend. Förbättringsåtgärderna hade varit att införa ännu ett fält som ger en tydlig beskrivning för vilken stegtyp det är. Ett sådant stegtyp är 'scope' som borde ha ett beskrivande fält som berättar vad för typ av scope det är. Om det är en rekursiv körning av en funktion eller om det är en vanlig kodblock i en funktion som körs exempelvis.

5.5.1 Behov av vidare kunskap

Det har blivit tydligt efter arbetet att flera av de tekniska områdena gruppen behandlat kräver en fördjupning för att kunna skapa ett mer komplett och robust verktyg. Mycket av det behovet är en grundlig förståelse för effektiv datastruktursoch algoritmdesign. För att skapa bättre logik för de nuvarande datastrukturerna, samt de som inte än blivit implementerade, krävs det mycket mer än bara en ytlig förståelse för deras användning.

Det krävs även en djup förståelse över hur datastrukturerna modelleras och manipuleras. I samband med det behövs även djupare kunskaper i hur proxys fungerar, så att loggningen kan utvecklas på så sätt att det påverkar algoritmernas kod så lite som möjligt. För att visualisering av parallell exekvering är ett naturligt nästa steg, hade det varit av vikt att sätta sig in i mekanismerna bakom parallell programmering. Dessa är bland annat event loops, multithreading och actor models.

Slutligen hade det varit av fördel med mer kunskap om användarcentrerad design och UX. Detta för att kunna designa en frontend som visualiserar algoritmerna på ett pedagogiskt och lättbegripligt sätt, särskilt inom utbildning.

5.5.2 Framtida problemställningar

Hur kan man visualisera parallell exekvering av algoritmer på ett sätt som är pedagogiskt och intuitivt för studenter?

Kan man skapa ett system där användare laddar upp kod och får en korrekt logg/visualisering automatiskt, med minimal manuell insats?

6

Slutsats

Arbetet utgick från en övergripande frågeställning om hur ett loggbaserat eDSL kan utformas för att möjliggöra visualisering av algoritmers beteende, utan att koppla samman exekveringslogik och presentation. De tre delmålen fungerade som konkretiseringar av denna frågeställning. För att uppnå detta skapades tre delmål: att utveckla ett DSL som kan representera stegen i exekveringen hos en algoritm, ett bibliotek som genererar denna logg med låg påverkan på algoritmers originalstruktur, samt att undersöka hur loggen kan användas för visuell återgivning.

Resultatet blev ett TypeScript-baserat backend-bibliotek som genererar loggar i form av ett JSON-baserat DSL. Detta DSL beskriver de enskilda operationerna, samt de mer omfattande grupperna av steg som en algoritm utfört, exempelvis block och funktioner. Alltså uppfylldes målet att utveckla ett DSL som representerar stegen hos en algoritms exekvering.

Biblioteket erbjuder färdiga datastrukturer som har loggningsfunktionalitet, vilket gör det möjligt att skriva loggbar kod med viss avvikelse från den originella koden för algoritmen. Detta uppfyller bara delvis målet att utforma loggningen så att den har minimal påverkan på den ursprungliga algoritmens kod, då avvikelsen hos koden som uppstår vid loggning blev större än önskat.

Utvecklandet av ett användargränssnitt för att visualisera loggen påbörjades, men det blev inte färdigställt i den utsträckningen som planerats. En mindre prototyp togs fram, men på grund av tidsbegränsning och prioritering av de andra delarna i arbetet fick det inte tillräckligt med stöd.

Trots detta visar arbetet på att separation mellan exekveringslogik och presentation, med hjälp av ett DSL, är möjligt. Det finns dessutom goda förutsättningar för vidare utveckling, både genom att öka antalet stödda datastrukturer och vidareutveckla visualiseringen, med ett pedagogiskt och användarcentrerat perspektiv.

Dessutom visar systemets uppbyggnad att det finns möjlighet att i framtiden utöka stödet till fler programmeringsspråk än TypeScript. Det finns också potential att underlätta integrering av loggning i redan existerande kod, utan att behöva skriva om algoritmerna i sig. På längre sikt skulle verktyget kunna komma till nytta både i utbildningsmiljöer och i praktiska felsökningssituationer där insyn i algoritmers beteende är värdefullt.

Referenser

[1] Alvilang contributors. ”alvilang-ts.” (2025), URL: https://github.com/alvilang/alvilang-ts (hämtad 2025-05-19).

[2] L. Bassett. ”Introduction to JavaScript Object Notation.” (2015).

[3] Chalmers tekniska högskola. ”Datastrukturer och algoritmer - Kursplan för DAT038.” (2025), URL: https://www.chalmers.se/utbildning/dina-studier/hitta-kurs-och-programplaner/kursplaner/DAT038/?acYear=2024%2F2025 (hämtad 2025-02-13).

[4] David Galles. ”Data Structure Visualizations.” (2025), URL: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html (hämtad 2025-05-19).

[5] Douglas Crockford. ”Introducing JSON.” (2024), URL: https://www.json.org/ (hämtad 2025-05-19).

[6] Eli Finkel. ”deepdiff – Deep Difference and Structural Comparison for Python.” (2025), URL: https://github.com/seperman/deepdiff (hämtad 2025-05-19).

[7] Erich Gamma, Richard Helm, Ralph Johnsosn och John Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 2009.

[8] P. Huldak, ”Domain Specific Languages,” Yale University, tekn. rapport, 1997.

[9] Jakub (kuko) Kováč, Katka Kotrlová, Pavol (paly) Lukča, Viktor (friker) Tomkovič och Tatiana Tóthová. ”Gnarley trees.” (2018), URL: https://kubokovac.eu/gnarley-trees/#.

[10] Juan Lin och Hui Zhang. ”Data Structure Visualization on the Web.” (2020), URL: https://ieeexplore.ieee.org/abstract/document/9378249 (hämtad 2025-02-28).

[11] Len Bass, Paul Clements och Rick Kazman, *Software Architecture in Practice, Third Edition*. Pearson Education, Inc, 2015.

[12] Markus Voelter, Sebastian Benz, Christian Dietrich m. fl. ”DSL Engineering.” (2013).

[13] Martin Fowler och Rebecca Parsons, *Domain-Specific Languages*. Pearson Education, Inc, 2012.

[14] MDN Web Docs, *Arrow function expressions*, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (hämtad 2025-05-27).

[15] MDN Web Docs, *Proxy - JavaScript | MDN*, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy (hämtad 2025-05-27).

[16] Michel Weststrate. ”Immer – Create the next immutable state by mutating the current one.” (2025), URL: https://immerjs.github.io/immer/ (hämtad 2025-05-19).
[17] Monika Akbar, Alexander Joel D. Alon, Matthew L. Cooper m. fl. ”Algorithm Visualization: The State of the Field.” (2010), URL: https://www.researchgate.net/publication/220094605_Algorithm_Visualization_The_State_of_the_Field (hämtad 2025-03-01).
[18] Mozilla Developer Network. ”JavaScript Documentation.” (2025), URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (hämtad 2025-03-02).
[19] Peter Ljunglöf. ”dsvis.” (2025), URL: https://github.com/ChalmersGU-data-structure-courses/dsvis (hämtad 2025-02-11).
[20] Python Software Foundation, *Python: Programming Language*, version 3.12, 2024. URL: https://www.python.org (hämtad 2025-05-24).
[21] Robert C. Martin. ”Agile Software Development, Principles, Patterns, and Practices.” (2003).
[22] A. Shvets. ”Design Patterns.” (), URL: https://refactoring.guru/design-patterns (hämtad 2025-04-14).
[23] TypeScript Team. ”TypeScript.” (2025), URL: https://www.typescriptlang.org/ (hämtad 2025-03-02).
[24] Wikipedia contributors. ”Domain-specific language — Wikipedia, The Free Encyclopedia.” (2024), URL: https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=1264507543 (hämtad 2025-02-11).
[25] Wikipedia contributors, *Sample sort*, 2024. URL: https://en.wikipedia.org/wiki/Samplesort (hämtad 2025-05-28).
[26] Wikipedia contributors. ”Serialization Wikipedia, The Free Encyclopedia.” (2024), URL: https://en.wikipedia.org/wiki/Serialization (hämtad 2025-05-19).
[27] Wikipedia contributors, *Systems architecture — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Systems_architecture&oldid=1289991007, [Online; accessed 19-May-2025], 2025.
[28] VisuAlgo Team. ”VisuAlgo – Visualising data structures and algorithms through animation.” (2025), URL: https://visualgo.net/en (hämtad 2025-05-19).