Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Development of an API for temporal coordination of multiple Services

Oliver Jackson

Course of Study: Informatik

Examiner: Prof. Dr. Marco Aiello

Commenced: March 31, 2020

Completed: November 23, 2020

Abstract

Personal Assistants like Alexa, Cortana or Siri have a lot of applications; however, many require special APIs to solve the posed problems. This Bachelor Thesis formally defines the problems FULLMATCH and FLEXMATCH, which require coordination of services on a temporal level. The developed API, named 'TempCo', solves these problems and its components are described and explained here in detail. It leverages the Semantic Web through JSON-LD and can parse data from various sources by using the Schema.org vocabulary. All algorithms were written in TypeScript which allows the code to be portable and run on many platforms. A standalone Node.js HTTP REST server implementation is also provided that listens to POST requests, enabling algorithm usage through the network. We analyse all algorithms on Performance to identify practical limits and runtime complexities. Results show that parsing JSON-LD objects can be slow for real time environments and the FLEXMATCH algorithm best works for small or constrained inputs. In contrast, the FULLMATCH algorithm is very scalable and can solve typical problem instances in a matter of seconds. Finally, we describe possible use cases, showing the versatility and applications of both algorithms to enhance Personal Assistants.

Contents

List of Figures

List of Listings

1 Introduction

In recent years multiple technologies have entered the marked and revolutionized how we view and interact with electronic devices. Technology continues to integrate into our lives, causing a shift in focus for electronic devices and applications, to not only work but also be easily accessible and usable. To achieve this, researchers have further explored the possibilities of using Artificial Intelligence in everyday life. This has lead to multiple advancements in Big Data such as: Autonomous Driving [ANM16] and Natural Language Processing (NLP) [AAB+15]. With NLP Personal Assistants such as Alexa, Cortana and Siri are able to fulfil a wide range of voice commands, from simple tasks like setting a reminder to complex actions such as finding a restaurant and reserving a table. However, there are still a lot of limiting factors when it comes to fulfilling the vast amount of possible user requests: first, each request has to be interpreted and converted from simple speech to something a computer can understand via NLP. Then all necessary services need to be identified, setting the order of query operations such that, for example, a suitable restaurant is found first before trying to reserve a table there. This step also requires spatial and temporal reasoning to ensure that results are plausible and relevant to the user. After the results are collected, a response is given to the user via Voice Synthesis where adjustments or confirmation is possible. Aside from the common problem of correctly interpreting the voice request, because NLP is still a complex problem, querying all necessary services continues to be the major hurdle Personal Assistants have yet to overcome. The main issue lies with third party services, that either don't provide easy access to APIs or flat out don't have any machine-readable data aside from webpages, where it's not trivial to extract the relevant information. Furthermore, each API requires effort from the developer to find available methods and integrating them into the application. APIs have to be documented, maintained and can introduce breaking changes which requires adjustments in each application that makes use of them. Adding to all this, if a new data source is introduced all steps for API integration have to be repeated which leads to a lot of work just maintaining APIs. Here Linked Data provides a way to publish data which is both accessible to humans and machine-readable. Just like linking documents on the Web with hyperlinks, Linked Data links things together, providing meaning and context. Combined with common, reusable vocabularies it enables developers to draw data from multiple sources without having to rely on APIs.

This Bachelor Thesis aims to improve temporal reasoning within Personal Assistants by developing an API that combines multiple services. In this context a service refers not just to web services but also to anything that offers a service in the real world such as reservations, bookings and resource management. Taking advantage of Linked Data through JSON-LD it's

possible to use the API in a multitude of domains without rearranging the required data. This makes it a valuable tool for dealing with user requests that can't be satisfied by one service alone but rather require orchestration of multiple services.

Outline

This Bachelor Thesis is structured in the following way:

Chapter 2 – API Specification Each component of the API and their interactions are specified. The basic communication, input and output data formats are defined.

Chapter 3 – Matching Algorithms Overview and explanation of the algorithms used for matching services to the user request.

Chapter 4 – Performance & Optimizations Analyses the API performance and reviews the results. Practical limits and runtime complexities are identified.

Chapter 5 – Use Cases Details possible use cases for the API and describes how it could be integrated in various domains.

Chapter 6 – Conclustion and Outlook Sums up the results of this thesis and gives an outlook for future research and development.

1.1 State of the Art

It is difficult to place TempCo in a specific research category, as it combines methods from multiple fields to achieve the desired result. To my knowledge, this exact application has not been researched and developed yet, however some subcomponents bear resemblance to other publications: when looking at the FULLMATCH problem defined in the next chapter, it is comparable to a job scheduling problem in which start and end times are fixed. First researched by Arkin et al. [AS87] where they presented an $\mathcal{O}(n^{k+1})$ algorithm for scheduling on k machines and relied mainly on graph theory. Kovaloyov et al. later reviewed similar problems with algorithms [KNC07] none of witch match the requirements imposed by this thesis. Mainly the introduction of a maximum wait time between intervals, limitation of a demand window and balancing of attributes make this problem unique. Weights are assigned to scheduled jobs which signify their value when processed by a machine but the goal is just to find the maximum weight for the given set of jobs and not something more complex such as balancing multiple factors based on user preference. Although other objectives have been explored [WBH20], they generally rely on a single fixed goal that characterizes the solution instead of factors given by the input.

The algorithms presented here rely on Allen's Interval Algebra [All83] and use fixed time points to define temporal intervals. From the available set of Allen's Relations only necessary ones are used and inferred for performance reasons.

Considering the second problem FLEXMATCH introduced here, it is comparable to the much researched Traveling Salesman Problem (TSP) with further alterations: Firstly time windows are introduced where each point can be visited and visits aren't considered instantaneous, they require a general service time that has to fit within the allowed time window. Furthermore, points are organized in groups that represent alternatives and each group has to be visited only once without a round trip. Finally, travel time is not a factor in this setting because points of interest are assumed to be relatively close to each other such that travel times can be negligible. Again to my knowledge this specific combination of features hasn't been researched yet but for each alteration there are multiple publications which present solutions: TSP with Time Windows(TSPTW) has seen exact [DDGS95, FLM02] as well as heuristic solutions [OT07, SU10]. R. F. Da Silva and S. Urrutia also maintain an online resource for comparing problem instances and solutions [SU17]. In general TSPTW is viewed as a delivery problem in which an agent needs to deliver goods to customers that has clear economic relations to resource distribution and online shopping [BDR20]. For these settings the time it takes to handle each customer is negligible and therefore not included in the problem description. However, there are instances in which a service time is significant [AV15] e.g. when waiting, processing or general browsing hence it's considered a factor for FLEXMATCH.

Group TSP or also referred to as the Errand Scheduling Problem considers the case where an agent needs to visit one point from all distinct groups in the shortest tour possible. This problem has also seen coverage through multiple papers which offer approximation algorithms [Sla97, EFMS05]. Exact methods also exist [Gar20] but are generally restricted to smaller problem instances because of the NP-hard TSP nature. Any correlation to time windows which could occur through opening hours or unavailable times are overlooked in the existing research.

Comparing TSP to FLEXMATCH is still valid even though the removal of travel times would make standard TSP trivial. The introduction of time windows with service time alone makes the problem much harder to solve. It is possible to represent time windows through an interval graph and if we were to repurpose service times as time on edges to other available locations a TSP instance would emerge with additional restrictions such as arriving too late at a location. Personal preferences also play an essential role within the FLEXMATCH algorithm which affect the outcome and introduce further complexity not considered in previous research. This thesis introduces FLEXMATCH as a novel problem and not through an extension of TSP to avoid potential confusion.

2 API Specification

The main goal of the API, henceforth called Temporal Coordinator (TempCo), is coordinating services to a given user demand. We formally define the problem FULLMATCH:

Let D be a demand with defined start- D_s and end-time D_e as well as a set O which contains offers o that have a start- o_s and end-time o_e. Find the best order of offers that closely match the given demand time window $[D_s, D_e]$ and don't contain delays between offers larger than a given value D_{md}. Whenever there is a trade-off between multiple attributes such as offer cost o_c, rating o_r or wait time between two offers $O[x]_s - O[y]_e$, the best solution is defined through user preference which attribute is valued over the others. This requires all attributes to have an assigned weight w_a with $0 \leq w_a \leq 1$ and $\sum_w = 1$ which signifies their relative importance.

We also define FLEXMATCH, an extension of this problem: Instead of just one set of offers there are now multiple services $s \in S$, which each contain a set of offers s_o. Furthermore, a new attribute service time o_t is introduced that describes how much time it takes to use that specific offer. Start- o_s and end-time o_e now form the boundaries in which an offer can be used. The time t at which an offer is utilized must allow for the service time to be included such that: $t + o_t \leq o_e$. Find the best order of service offers in S which utilizes at least one offer from each service within the demand window $[D_s, D_e]$.

For TempCo to be deployed in a multitude of scenarios and applications, we need it to be portable and easily integrable. Additionally, it should be possible to deploy the API as a web service or just include the module. Any language which needs to be compiled such as C++, Go, Rust will always run into the issue of platform compatibility and dependency, which complicates development and might cause further problems. Choosing an interpreted language comes with its own sets of limitations mainly platform availability and performance. However, we've come a long way since the first interpreted languages and interpreters for popular ones like Python, Java and JavaScript exist on all relevant platforms while performance increases through methods such as just-in-time compilation. Ultimately Node.js was chosen, due to its popularity and versatility, fitting the described needs for platform compatibility and options for deployment. Node.js is a JavaScript runtime built on the V8 Google high-performance JavaScript engine [Joy20], it allows for applications written in JavaScript to run as a server application and some components can also run inside any modern browser such as Google Chrome or Mozilla Firefox with minimal adjustments. This enables more potential use cases for TempCo such as directly embedding specific components into a website, eliminating the need for server communications and granting the ability to run any algorithms directly on the

user device. However, even if JavaScript is ubiquitous in modern Browsers, implementations of the core engine can vary greatly which can cause issues when developing applications. The European Computer Manufacturers Association (ECMA) takes care of standardizing JavaScript features, which are part of the ECMAScript that each major browser manufacturer gradually implements. Because distributing and updating these newer versions of browser takes time, every user might have a different JavaScript environment running, which means developers can't rely and utilize the newest functions natively. This lead to the choice of TypeScript, a typed superset of JavaScript created by Microsoft [Mic20], as the final development language.

TypeScript must be compiled to JavaScript before it can be run by Node.js or any browser, which enables the compiler to compile application code for a specific ECMAScript standard. New features can be used in the code which, when not available in the target version, will get translated by the compiler using older functions. Furthermore, TypeScript offers static typing, that enables the IDE to identify variables, show potential class methods and type check code for errors. While this works flawlessly for your own code, TypeScript relies heavily on type definition files for external JavaScript libraries. If these are not provided by the original creator, the community can provide custom type definition at the risk of them being wrong, incomplete or outdated. Although it's still possible to use untyped JavaScript libraries, TypeScript won't be able to provide any static code checking capabilities relating to these libraries. However, for this project, almost all used libraries had up-to-date definition files readily available, which simplified integration and development speed was overall increased thanks to TypeScript.

2.1 Component overview

TempCo is split into multiple components shown in Figure 2.1, these have specific purposes to logically split the workload between them. The next sections present each component: Request Handler, Matcher and Parser, specifying their interactions and details how they are implemented. A large part of the API is dedicated to convert the user request data into objects for the matching algorithms. If needed, Parser and Matcher can be used independently in other applications. Objects required by the Matcher can also be instantiated separately to avoid data de-serialization.

2.2 Request Handler

If operating in server mode the Request Handler is responsible for processing all requests to TempCo from users. It uses the 'hapi Framework', developed by Sideway and originally intended to handle large scale demands [Sid20], to implement a REST HTTP server which operates specific routes corresponding to the different problems. Listing 2.1 shows the general structure of all TempCo routes: Each route defines a path on which the handler function is

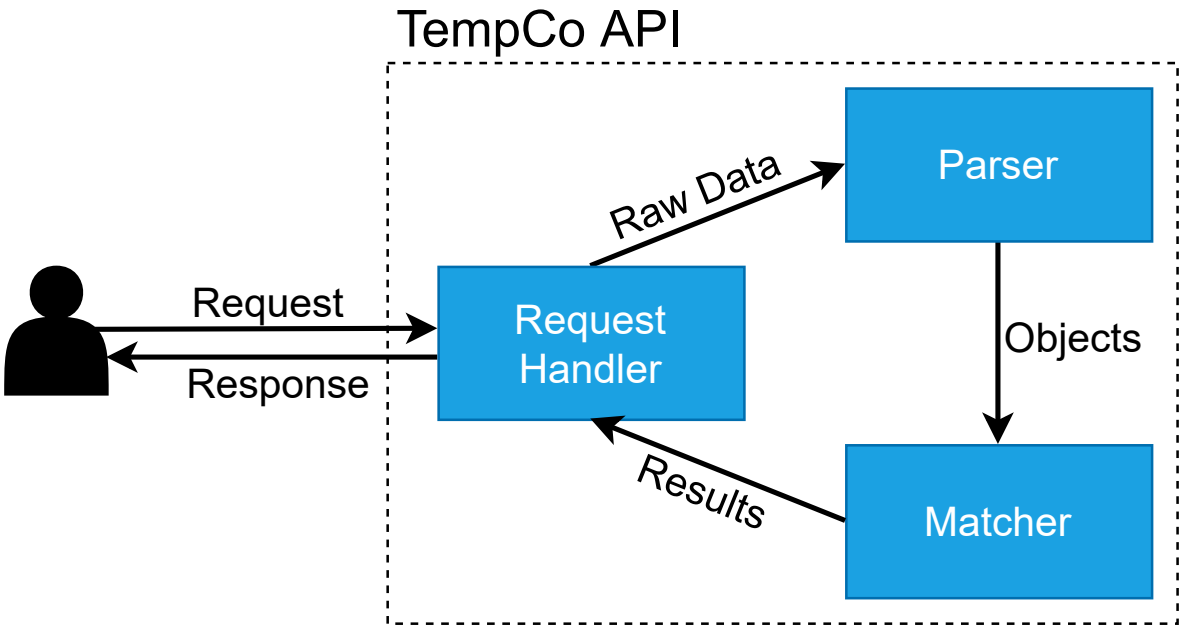


Figure 2.1: Specification of components and their interactions in TempCo.

executed. We exclusively use the HTTP Post method for all requests, which should contain the algorithm data in the post body (payload) as JavaScript Object Notation (JSON).

The parser expects serialized data in JSON; however, because we want to utilize the semantic web we're using an extension of JSON called 'JSON for Linked Data' (JSON-LD). JSON-LD is one of many ways to store linked data and is itself an extension of the Resource Description Framework (RDF). Demand data is given through a static structure but to facilitate using data from different sources and services, offers have to be provided in JSON-LD which is further explained in section 2.3. Additionally, we use the 'Joi' module to validate the basic structure and types of the data. Joi allows us to create schemas within `options.validate` which automatically validates and parses all incoming data before it reaches the handler function. This allows us to write code relying on all required parameters to exist and them being the correct type. Because all TempCo algorithms are dependent on different data each route reflects this:

/fullmatch Is the main endpoint for all FULLMATCH queries. This algorithm uses offer ratings and price when calculating the best offer matching for a given demand timeframe. This requires weight values to be defined for each attribute, which is why they are required in the request.

/fullmatch/simple Accesses a simpler but faster FULLMATCH algorithm which only aims to match the maximum coverage of offers within the given demand, ignoring other factors such as cost or rating.

/flexmatch Solves FLEXMATCH problems and requires the same demand parameters as **/fullmatch**. However, instead of the `data` parameter it expects a `services` array in the post body which wraps offer data, so they can be correctly assigned to each service.

Furthermore, each route accepts a query parameter `skip_invalid` that when present, tells the parser to not stop parsing if a incomplete or invalid offer is detected. It should be used when provided data can be incomplete but the algorithms should still try to work with whatever data is usable.

The handler functions all work similarly as shown in Listing 2.2 for the FULLMATCH handler function. First the parser is called to parse all required data (line 5), depending on if any errors occur, we either use the 'Boom' library to generate an HTML friendly error object which contains information about the `ParseError` (8) or we further throw the unexpected error (10) to be logged and inspected. Afterwards a new `Matcher` gets instantiated with parsed `demand` (12) and the appropriate algorithm is called (13). If no matching is possible `result` will be an `MatchingError` which contains further information that can be relayed to the user (14-16). Otherwise, we return the results in JSON-LD format (17-22) which gets send as the response to the user.

Listing 2.1 Route configuration for the simple FULLMATCH algorithm

```
1  server.route({
2      method: 'POST',
3      path:'/fullmatch',
4      handler: async (request, h) => // handler function
5      options: {
6        validate: {
7          payload: Joi.object({
8            demand: Joi.object({
9              start: Joi.string().isoDate().required(),
10             end: Joi.string().isoDate().required(),
11             weights: Joi.object({
12               coverage: Joi.number().required(),
13               rating: Joi.number().required(),
14               cost: Joi.number().required()
15             }).required(),
16             maxDelay: Joi.number().min(0)
17           }),
18           data: Joi.alternatives().try(Joi.object(), Joi.array()).required()
19         })
20       }
21     }
```

Listing 2.2 /fullmatch route handler function

```
async (request, h) => {
  let demand;
  let skipInvalid = request.query.skip_invalid ? true : false;
  try {
    demand = await Parser.parseFullmatch(request.payload, skipInvalid);
  } catch (error) {
    if (error instanceof ParseError) {
      return Boom.badRequest(error.message);
    }
    throw error;
  }
  let matcher = new Matcher(demand);
  let result = matcher.fullmatch();
  if (result instanceof MatchingError) {
    return Boom.badData(result.message);
  }
  return {
    "@context": {
      "@vocab": "http://schema.org/"
    },
    "@graph": result.map((offer) => offer.toJSONLD())
  }
}
```

2.3 Parser

The parser is responsible for parsing raw JSON data into objects used by the matcher. These include the following classes: `Service`, `ServiceOffer` and `UserDemand` shown in Figure 2.2.

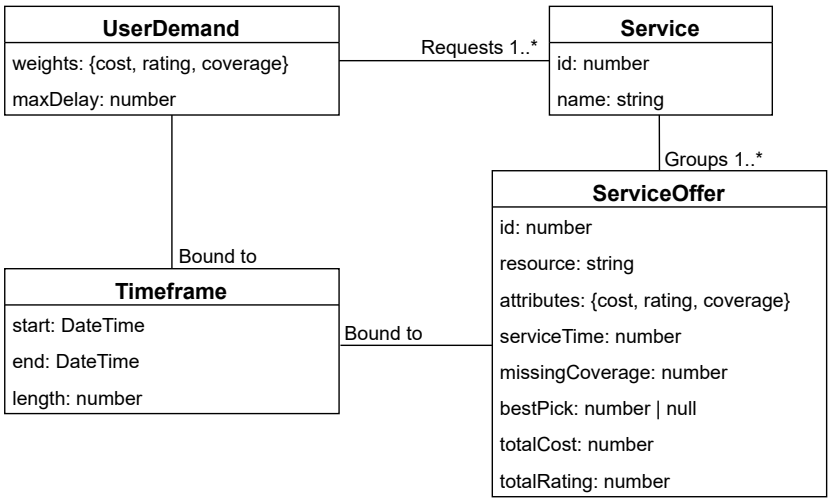


Figure 2.2: UML diagram of parsed classes used for the matching algorithms.

UserDemand This class contains all relevant data about a specific user demand. This includes the weights for each attribute representing the user preference and a value `maxDelay` which describes the maximum allowed time delay between two matched offers so that the matching is still valid. It is also bound to a `Timeframe` giving the Matcher temporal boundaries where a match can occur. All provided offers or services are also stored here for easy access by the Matcher.

Service This class describes a specific service requested by the user. The `name` attribute is used to store the reference to the real world service for the response. Possible services could include a business where the user wants to shop but also specific rooms inside the user's house where a service, such as cleaning or ventilation is needed. Furthermore, the service object acts as a container for `ServiceOffers`, which are grouped to signify that each `ServiceOffer` offers an equivalent service, and they can be used interchangeably.

ServiceOffer Here, the attributes for a concrete service offering are stored. These can be extended to meet the demands of a specific domain but mainly consist of cost, rating and wait time. A `Timeframe` object within describes when and how long a `ServiceOffer` is available by storing the start and endpoint as a `DateTime` object. Each timeframe has to be continuous and any discontinuities are represented by creating multiple `ServiceOffers` with their respective start and endpoints. Each `ServiceOffer` also has a `resource` to evaluate which occupy the same physical or logical entity and plays a role during matching. Internally each `Service` and `ServiceOffer` gets assigned an unique `id` to identify them in algorithms, even if their `Service.name` or `ServiceOffer.resource` is the same.

The `UserDemand` can be specified as shown in Listing 2.3. In order to obtain all required values for the algorithms we're utilizing JSON-LD, which adds semantic context to JSON documents in the form of a special JSON key called `@context`. Listing 2.4 shows how the context is used to define a common vocabulary for all keys within that document and some key aliases.

This example utilizes Schema.org, a collection of schemas for structured data on the Internet. It is funded and used by major tech companies such as Google, Microsoft, Yahoo and Yandex but also maintained by a larger web community [sch20]. When processing JSON-LD, each

Listing 2.3 Specification of `UserDemand` within the request payload

```
1  "demand": {
2    "start": "2020-03-31T14:00:00Z",
3    "end": "2019-03-31T20:00:00Z",
4    "weights": {
5      "coverage": 0.2,
6      "rating": 0.3,
7      "cost": 0.5
8    },
9    "maxDelay": 3600000
10 }
```

Listing 2.4 JSON-LD example for `ServiceOffer` with `@context`

```
"@context": {
  "@vocab": "http://schema.org/",
  "start": "availabilityStarts",
  "end": "availabilityEnds" },
"@graph": [{
  "url": "http://example.com/Room1",
  "start": "2020-03-31T14:00:00Z",
  "end": "2020-03-31T16:00:00Z",
  "aggregateRating": 4.5,
  "price": 100
},{
  "url": "http://example.com/Room2",
  "start": "2020-03-31T16:00:00Z",
  "end": "2020-03-31T18:00:00Z",
  "aggregateRating": 3,
  "price": 50
}]
```

key will be extended to an 'Internationalized Resource Identifier' (IRI) which points to the definition for that item type or property, e.g. `price` gets extended to `http://schema.org/price`. The parse can utilize this information together with another concept of JSON-LD called 'Framing': Framing allows us to restructure JSON-LD in a consistent, expected way through the use of a frame, these enable the parser to retrieve values through semantics rather than specific JSON structures. With the frame shown in Listing 2.5, we define which context is used, the properties we are interested in: `resource`, `availabilityStarts`, `availabilityEnds`, `aggregateRating` and `price` as well as framing flags such as `@explicit`, which sets if properties that are not present in the frame should be included in the result.

Listing 2.5 JSON-LD Frame used for parsing `ServiceOffer`

```
"@context": {
  "@vocab": "http://schema.org/",
  "resource": "url" },
"@explicit": true,
"resource": {},
"availabilityStarts": {},
"availabilityEnds": {},
"price": {},
"aggregateRating": {}
```

This provides us with the structure shown in Listing 2.6 where we can see all framed properties within keys in `@graph`. Notice how the aliased property keys `start` and `end` are now using the standard vocabulary keys from Schema.org and `url` is accessible with the key `resource`. However, there's a caveat when just parsing each `@graph` object as a `ServiceOffer` object: duplicate or nested properties. By using the frame in Listing 2.5, we get objects with properties that are

on the same nesting level, if there's a property anywhere else within the data, then the result graph would contain another object with this property and non-present set to `null`. In order to prevent this, the parser iterates over all elements in the graph, if the current element is an object the context will be inserted, otherwise if it's an array a new graph is created with the array elements and the context attached. Now each element can be framed individually and resulting properties are combined to form the final result object, which is then parsed to a `ServiceOffer` object. One major benefit of framing individual objects this way, is that any properties which are not present in the original data can be easily fetched from other sources, such as review aggregation sites and just append them to the data array for that offer. During the combination step these will be merged into the final `ServiceOffer` object.

Data and structure which must be provided is dependent on the problem:

FULLMATCH All FULLMATCH offers essentially belong to the same class of services, which is why they can be directly provided within a request by using the `data` key. This must be either an object with a `@context` and `@graph` or an array of arrays or objects containing the necessary offer data, which must include: An unique `url` identifier for the physical or logical resource this offer occupies, a start time in `availabilityStarts`, an end time `availabilityEnds` and if matching should respect price or rating, a `price` or `aggregateRating` must be provided. Listing 2.4 shows an example on how this data can be structured.

FLEXMATCH Because FLEXMATCH works with multiple services that all have a unique set of offers, they need to follow a specific structure in order to correctly categorize them:

```
services: [
  {
    "name": "Service name"
    "data": ...//Offer data
  }
  ...
]
```

Each service needs a `name` which is used in the response to relay the matched order of services, offer `data` can be provided just like with FULLMATCH, except instead of `price` a `priceRange` is expected and the fixed timeframe from `availabilityStarts` to `availabilityEnds` can be replaced an `openingHoursSpecification`. This details for every day of the week at which times an offer is available and each timeframe within the demand window gets converted to a separate `ServiceOffer` for matching. The service time, that states how long an offer must be utilized, is given through an additional property `eligibleDuration`. It allows for flexible duration definition with a UN/CEFACT `unitCode` [UNE20] that gives the attached `value` magnitude.

Listing 2.6 Result of framing Listing 2.4 with the frame in Listing 2.5

```
"@context": {
  "@vocab": "http://schema.org/",
  "resource": "url" },
"@graph": [{
  "aggregateRating": 4.5,
  "availabilityEnds": "2020-03-31T16:00:00Z",
  "availabilityStarts": "2020-03-31T14:00:00Z",
  "price": 100,
  "resource": "http://example.com/Room1"
},{
  "aggregateRating": 3,
  "availabilityEnds": "2020-03-31T18:00:00Z",
  "availabilityStarts": "2020-03-31T16:00:00Z",
  "price": 50,
  "resource": "http://example.com/Room2"
}]
```

2.4 Matcher

Once all objects have been parsed, the Matcher can be instantiated which is responsible for solving the actual FULLMATCH and FLEXMATCH problem instances. It offers methods to solve both problems exactly and returns the best found result, or a `MatchingError` with further information on why no matching is possible. Objects as shown in Figure 2.2 are designed to store certain values used by the algorithms, while larger data structures are stored and exposed in the Matcher.

3 Matching Algorithms

In this chapter the algorithms to solve FULLMATCH and FLEXMATCH problems are detailed, showing relevant implementation details whenever necessary.

3.1 FULLMATCH Algorithm

FULLMATCH is closely related to a weighted scheduling problem; however, instead of being able to string together different offers o, we need to respect their temporal boundaries within the demand boundaries. A valid result for this problem is a ordered list L with length n, containing only offers where no wait time exceeds the maximum allowed delay D_{md}:

$$
\begin{aligned}
L[1]_s - D_s &\leq D_{md} && \text{Demand start to first offer} \\
D_e - L[n]_e &\leq D_{md} && \text{Last offer to demand end} \\
\forall_{1 \leq i < n}\ L[i+1]_e - L[i]_s &\leq D_{md} && \text{Between any two adjacient offers}
\end{aligned}
$$

Each possible arrangement for L has to be checked theoretically to find the best overall solution. However, we can utilize that time is only moving forward, meaning for each offer o we don't need to consider any overlapping or earlier offers. This leads us to a bottom-up algorithm, where we first order all offers by their start time o_s, calculate the best solution for the last offer and then gradually include each previous offer until we end up with the solution for all offers. Listing 3.1 shows the code for the simple FULLMATCH algorithm.

For each offer we keep track of the best possible next offer index with `offer.bestPick` which forms a linked list. The best offer is first determined by the lowest value when calculating the duration which is not covered between $[D_s, D_e]$ by this offer and all subsequent `bestPick`. We store this value in `offer.missingCoverage`, iterating backwards through the offers initializing `offer.missingCoverage` (line 5 - 7) with the temporal distance to demand start and end: $o_s - D_s + D_e - o_e$. Function `calcDistance(start, end)` ensures that any temporal distance is capped at 0 to prevent potential underflows when offer timeframes exceed the demand window. Next, we iterate from the current offer to the list end, skipping all offer pairs which timeframes overlap (10) because we can't pick two offers simultaneously. For each valid pair we check if choosing the selected offer as `bestPick` would decrease the current best `missingCoverage` (34 - 35). If it does, we update `missingCoverage` and `bestPick` accordingly (36 & 37). This also takes

into account the `sameResourceBonus`, which is a primitive way of checking if a resource can be occupied between two adjacent offers. Assuming these have the same `resource` and the time between them is not higher than one offer length, then it should be possible to combine the offers. Which allows us to model interactions like staying multiple nights in an hotel, using a room throughout scheduling blocks and favour offers that don't require the user to switch. If the conditions are met, `sameResourceBonus` gets set to the distance between both offers and subtracted from the `selected.missingCoverage`.

Lines 28 - 33 handle cases where `combinedCoverage` could potentially underflow i.e. when offer timeframes exceed the demand window boundaries. If any offers go beyond the demand start or end, we simply subtract the distance from `timeframeLength`.

Furthermore, there are some optimizations we can do to reduce the average inner loop runtime: Line 13 checks if the distance from current offer to selected is greater than the maximum delay allowed by the demand, which means we can stop iterating because in the sorted offer list all following offers will have a distance $\geq$ than the selected. Variable `earliestBest` provides a more advanced optimization which keeps track of the earliest `bestPick` appearing in the inner loop. If any selected offer has a `bestPick`, it gets initialized to that offer first (14 - 16) before comparing it to the current `earliestBest`, updating it if the start time is lower (17 - 20). This variable allows us to stop iterating if we encounter the `earliestBest` as a selected offer in the inner loop (11).

Proof. Suppose there is an offer $O[x]$ after the `earliestBest` offer $O[b]$, which provides a lower total `missingCoverage` when picked. Having an `earliestBest` means there is an offer $O[m]$ which is between the current offer and $O[b]$. Because all offers are sorted, the start time of $O[x]$ is greater or equal to the offer $O[b]$. Choosing offer $O[x]$ as `bestPick` for the current offer would cause a temporal gap of at least the duration $O[m]$ meaning a higher overall `missingCoverage`. Furthermore, if offer $O[x]$ could provide a better coverage than $O[b]$, then $O[m]$ `bestPick` must be $O[x]$, which is pick able from $O[m]$ because $O[m]_e \leq O[b]_s \leq O[x]_s$ and was checked in an earlier iteration at line 35. □

Finding the best combination of offers requires us to keep track on the best starting index which is done through the matcher variable `bestStart`. It gets initialized to -1 and overwritten with the first valid offer in line 42. For an offer to be valid as a starting offer it just needs to have a temporal distance to the demand start which is $\leq$ than the maximum allowed delay (41). At the end of each outer loop iteration if `bestStart` is set we check if the current offer has a lower `missingCoverage` and update `bestStart` accordingly (44 - 46). If we can't find any compatible offers to the current one we set the `bestPick` to `null` (47 - 49), which signals that offer was processed and is an ending offer.

The final step is to follow the pointers from `bestStart` to each `bestPick` until we reach `null`. Listing 3.2 shows the matcher function implementation for following each pointer and returning the result list of selected offers. We first check if `bestStart` was initialized (1 - 3) and then begin following the `bestPick` index pointers in a while loop, adding them to the `selectedOffers`

Listing 3.1 Simple Fullmatch algorithm to minimize overall `missingCoverage`

```
let offers = this.prepareFullmatch();
for (let i = offers.length - 1; i >= 0; i--) {
  let offer = offers[i];
  let earliestBestPick = null;
  offer.missingCoverage =
    calcDistance(this.demand.start, offer.start) +
    calcDistance(offer.end, this.demand.end);
  for (let j = i + 1; j < offers.length; j++) {
    let selected = offers[j];
    if (offer.timeframe.overlaps(selected.timeframe)) { continue; }
    if (earliestBestPick === selected) { break; }
    let selectedDistance = calcDistance(offer.end, selected.start)
    if (selectedDistance > this.demand.maxDelay) { break; }
    if (selected.bestPick) {
      if (earliestBestPick === null) {
        earliestBestPick = selected.bestPick;
      } else {
        if (selected.bestPick.start < earliestBestPick.start) {
          earliestBestPick = selected.bestPick;
        }
      }
    }
    let timeframeLength = offer.timeframe.length;
    let sameResourceBonus = 0;
    if (offer.resource == selected.resource && selectedDistance < timeframeLength) {
      sameResourceBonus = selectedDistance;
    }
    if (offer.start < this.demand.start) {
      timeframeLength -= calcDistance(offer.start, this.demand.start);
    }
    if (offer.end > this.demand.end) {
      timeframeLength -= calcDistance(offer.end, this.demand.end);
    }
    let combinedCoverage = selected.missingCoverage - timeframeLength - sameResourceBonus;
    if (combinedCoverage < offer.missingCoverage) {
      offer.missingCoverage = combinedCoverage;
      offer.bestPick = selected;
    }
  }
  if (this.bestStart == -1) {
    if (calcDistance(this.demand.start, offer.start) <= this.demand.maxDelay) {
      this.bestStart = i;
    }
  } else if (offer.missingCoverage < offers[this.bestStart].missingCoverage) {
    this.bestStart = i;
  }
  if (offer.bestPick === undefined) {
    offer.bestPick = null;
  }
}
return this.getBestCombination(offers);
```

list until we reach `null` (6 - 9). Furthermore, we have to check if the last offer handled in the while loop is valid, meaning its temporal distance to the demand end is less than the maximum allowed delay (10 - 12). Finally, this function returns a list containing the optimal offers or a `MatchingError` if there is no valid solution.

Listing 3.2 Matcher `getBestSelection` function.

```
1  if (this.bestStart == -1) {
2    return new MatchingError("No suitable start offer found!");
3  }
4  let currentOffer = offers[this.bestStart];
5  let selectedOffers: ServiceOffer[] = [currentOffer];
6  while (currentOffer.bestPick) {
7    currentOffer = currentOffer.bestPick;
8    selectedOffers.push(currentOffer);
9  }
10 if (calcDistance(currentOffer.end, this.demand.end) > this.demand.maxDelay) {
11   return new MatchingError(‘Temporal distance from last available offer
         ’${currentOffer.resource}’ to demand end is too long!‘);
12 }
13 return selectedOffers;
```

3.2 Extending FULLMATCH with TOPSIS

While the algorithm in Listing 3.1 works for finding the maximum coverage within a given demand window $[D_s, D_e]$, in the real world we might want to consider trade-offs when it comes to certain attributes: for example, let there be two offers $O[x]$ and $O[y]$, $O[x]$ costs more than $O[y]$ but has a higher overall rating or $O[y]$ starts hours later than $O[x]$ but costs less, which one should be picked comes down to personal preferences. Is paying more acceptable for getting a higher quality service or are there budgetary constraints that influence a decision? Is a longer delay better than taking a lower rated offer? We can't have the user decide for each offer manually when there are possibly thousands of offers which need to be compared. This requires a 'Multiple Attribute Decision Making' (MADM) method to properly rank all available offers based on user preference. TempCo has certain requirements which this method must fulfil in order to be used by the matcher:

- It must be able to differentiate between attributes that have to be maximized such as ranking and minimized like cost and delays.
- The manner in which user preferences must be presented to the method should not be too involved and any required values have to be obtainable in an easy-to-understand fashion, such as asking the user for a ranking of attributes. This is mainly to facilitate using TempCo via a Personal Assistant through asking simple questions, which evaluate to preference values for the MADM method.

- Time complexity should not exceed $\mathcal{O}(n)$, because depending on the matching algorithm used, we might need to call the method many times, leading to increased response times. Especially with Personal Assistant quick results are a necessity, anything on top of processing time and network delays adds up rather quickly and makes the application seem unresponsive.

Based on these criteria there are a number of methods available as described by Tzeng, G.-H., & Huang, J.-J. [TH11]. For this project the 'Technique for Order Preference by Similarity to Ideal Solution' (TOPSIS) originally developed by Hwang, C.-L. & Yoon, K. in 1981 [HY81] was chosen, because it met all requirements and the basic method can be implemented rather easily. TOPSIS works on the concept that for each set of alternatives there's an ideal solution A^+ which has the best attribute values and an negative ideal solution A^- with the worst values from each alternative. The best alternative is chosen based on the shortest geometric distance to A^+ while having the longest to A^-. Figure 3.1 visualizes how the method works for two attributes and Listing 3.4 shows the implementation of TOPSIS rank calculation.

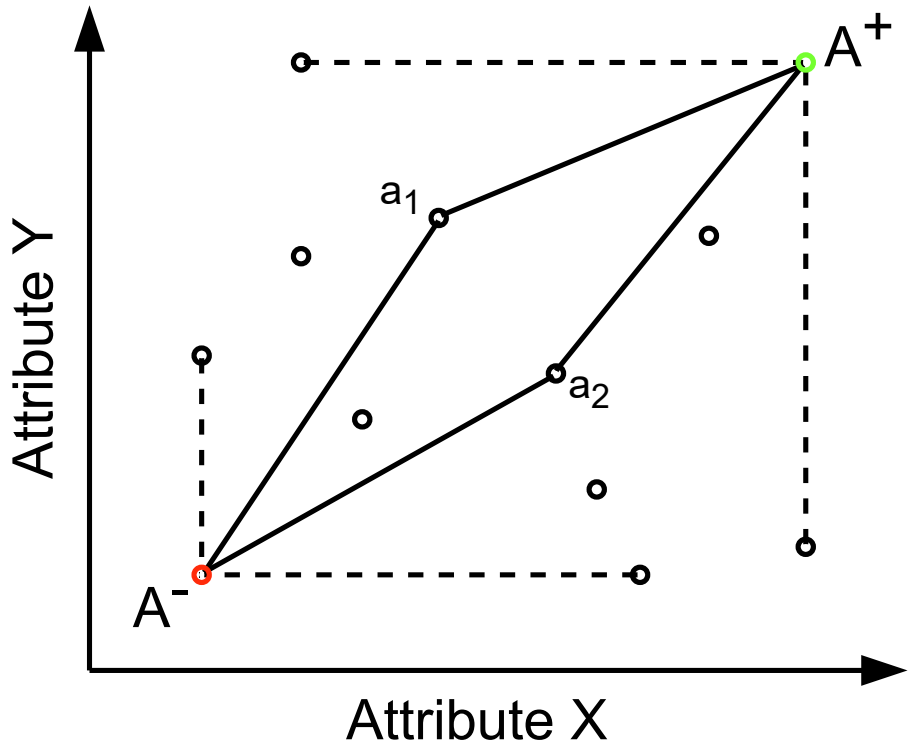


Figure 3.1: Concept of TOPSIS visualized with positive attributes: distance of alternatives a_1, a_2 to ideal solution A^+ and negative ideal solution A^-.

TempCo provides a specialized `Topsis` class, that offers functions to provide all required values for the 5 step rank calculation:

1. Find the best and worst value for each attribute respecting if it should be minimized or maximized and calculate the total for each attribute value squared. We can optimize this step by calculating minimum and maximum within the inner loop while iterating through the offers. All values can be provided to the `Topsis` instance by calling `include(attributes)` which will update the minimum or maximum value for each attribute in addition to calculating the totals as shown in Listing 3.3.
2. Calculate the norm for each attribute by taking the square root of the calculated total (Listing 3.4 line 4). We use this norm to normalize all attributes on a $[0, 1]$ scale which also applies to the ideal and negative ideal values (10 & 11). In case no attribute value is greater than zero the norm will equal 0, to prevent any division by zero errors and influence on the ranking, we set the norm to 1 and initialize the ideal and negative ideal values to 0 (5 - 8).
3. Determine the distance for each alternative to the ideal and anti-ideal value. We first normalize each attribute value (20) before summing the weighted ideal and negative ideal difference squared (21 & 22) to calculate the euclidean distance for both ideals (24 & 25).
4. Use the distances to calculate the final rating for each alternative (26) which gets inserted into the max heap (27).
5. After all ranks have been determined we can either use `extractRoot` on the max heap to get the best alternative or use heap sort to get an ordered list of all alternatives.

Listing 3.3 TOPSIS attribute include function

```
1  for (const name in attributes) {
2    let atr = this.attributes[name]!;
3    let value = attributes[name];
4    atr.total += value ** 2;
5    if (atr.maximize) {
6      if (value > atr.best) { atr.best = value; }
7      if (value < atr.worst) { atr.worst = value; }
8    } else {
9      if (value < atr.best) { atr.best = value; }
10     if (value > atr.worst) { atr.worst = value; }
11   }
12 }
```

Now we can modify Listing 3.1 to include the TOPSIS ranking method as shown in Listing 3.5. First we have to find the range of offers on which to apply the TOPSIS ranking, we do this by defining a `topsisStart` index which gets initialized in the inner loop with the first offer index that is not overlapping with the current offer and which distance is smaller than D_{md} (line 12 - 14). The upper bound is obtained through iteration variable `j` which is conveniently the last valid offer index non-inclusive. We do the same calculations as before, to get the combined

Listing 3.4 TOPSIS calculation function

```
let maxHeap = new MaxHeap();
for (const name in this.attributes) {
 let atr = this.attributes[name]!;
 atr.norm = atr.total ** 0.5;
 if (atr.norm == 0) {
  atr.norm = 1;
  atr.normBest = 0;
  atr.normWorst = 0;
 } else {
  atr.normBest = atr.best / atr.norm;
  atr.normWorst = atr.worst / atr.norm;
 }
}
for (let i = start; i < end; i++) {
 let offer = this.alternatives[i];
 let bestSeparation = 0;
 let worstSeparation = 0;
 for (const name in this.attributes) {
  let atr = this.attributes[name]!;
  let value = offer.attributes[name] / atr.norm;
  bestSeparation += ((atr.normBest - value) * atr.weight) ** 2;
  worstSeparation += ((atr.normWorst - value) * atr.weight) ** 2
 }
 bestSeparation = Math.sqrt(bestSeparation);
 worstSeparation = Math.sqrt(worstSeparation);
 let finalRating = worstSeparation / (bestSeparation + worstSeparation);
 maxHeap.insert(finalRating, i);
}
return maxHeap;
```

missing coverage (15-26) however, instead of saving it directly to `offer.missingCoverage` we first save it to the offers `attributes` object. These act like working memory that can get overwritten in each iteration and accessed by `Topsis` for rank calculation. Furthermore, for each available offer we need to calculate the totals for each attribute. We use the current total values from each selected offer, add the offer attribute value and save it in `attribute` (27 & 28). These totals are initialized when no suitable next offer is available (36 - 38) by using the offer attributes and the temporal distance to demand start and end. Using attribute totals allows us to find the global optimal solution because suboptimal offer attributes get propagated through the attribute total.

It is not possible to use the `earliestBestPick` optimization, because we can't simply look for the maximum coverage but also have to take attribute trade-offs into consideration. Any pick able offer could provide a better overall solution even if the coverage is worse. We also need to consider *trap* offers that have very good attributes but prevent us from picking anything else, if this offer is more than D_{md} away from the demand end, any `bestPick` pointing there would

Listing 3.5 Modification of Listing 3.1 to include the TOPSIS ranking method

```
let offers = this.prepareFullmatch();
for (let i = offers.length - 1; i >= 0; i--) {
  let offer = offers[i];
  let topsis = new Topsis<MatchAttributes>(new MatchAttributes(), offers, this.demand.weights);
  let j = i + 1;
  let topsisStart = -1;
  for (; j < offers.length; j++) {
    let selected = offers[j];
    if (offer.timeframe.overlaps(selected.timeframe)) { continue; }
    let selectedDistance = calcDistance(offer.end, selected.start);
    if (selectedDistance > this.demand.maxDelay) { break; }
    if (topsisStart == -1) {
      topsisStart = j;
    }
    let timeframeLength = offer.timeframe.length;
    let sameResourceBonus = 0;
    if (offer.resource == selected.resource && selectedDistance < timeframeLength) {
      sameResourceBonus = selectedDistance;
    }
    if (offer.start < this.demand.start) {
      timeframeLength -= calcDistance(offer.start, this.demand.start);
    }
    if (offer.end > this.demand.end) {
      timeframeLength -= calcDistance(offer.end, this.demand.end);
    }
    selected.attributes.coverage = selected.missingCoverage - timeframeLength - sameResourceBonus;
    selected.attributes.cost = selected.totalCost + offer.cost;
    selected.attributes.rating = selected.totalRating + offer.rating;
    topsis.include(selected.attributes);
  }
  if (topsisStart == -1) {
    if (calcDistance(offer.end, this.demand.end) > this.demand.maxDelay) {
      offers.splice(i, 1);
    }
    offer.bestPick = null;
    offer.missingCoverage = calcDistance(this.demand.start, offer.start) +
        calcDistance(offer.end, this.demand.end);
    offer.totalCost = offer.cost;
    offer.totalRating = offer.rating;
    continue;
  }
  offer.bestPick = offers[topsis.getBest(topsisStart, j)];
  offer.totalCost = offer.bestPick.attributes.cost;
  offer.totalRating = offer.bestPick.attributes.rating;
  offer.missingCoverage = offer.bestPick.attributes.coverage;
  if (calcDistance(this.demand.start, offer.start) <= this.demand.maxDelay) {
    this.possibleOptions.push(offer);
  }
}
this.findBestStart();
return this.getBestCombination(offers);
```

get trapped with an invalid solution. To prevent this we check for each offer that is an end offer i.e. has no `bestPick`, if it is valid (32) and remove any invalid end offers from the list (33). This guarantees us that every possible selection of offers can lead to a valid solution otherwise we end up with an empty list. Having found a non-empty range of offers for the TOPSIS ranking, we run the calculation on this range (41) and update the offer totals to reflect the `bestPick` choice (42-44). We can simply use the calculated attributes of the `bestPick` because they contain the current offer values calculated within the inner loop. Here we also check if an offer is a valid start offer i.e. its distance to the demand start is $< D_{md}$ and add it to the list of possible start options (45 - 47). After we finish iterating through all offers, we can use TOPSIS one more time to find the final `bestStart` offer index (49). Then we can finally follow the `bestStart` and `bestPick` pointers to obtain the final result just as before with Listing 3.2.

3.3 FLEXMATCH Algorithm

The FULLMATCH algorithms works only if the choice for an offer is either all or nothing, meaning you can't divide the offer timeframe into smaller sections based on specific needs. To solve a FLEXMATCH problem we need an entirely new algorithm because the approach and complexity are totally different from any FULLMATCH problem. Obtaining an exact solution for FULLMATCH essentially requires every combination of offers to be checked with regard to their services. Which means we easily hit time complexities in the realm of $\mathcal{O}(n!)$ if we can't implement any optimizations in our algorithm. Fortunately, we can analyse the data to prevent combinations that are strictly worse than others. Based on this analysis we only need to consider a smaller subset of combinations which saves computational time at the cost of additional memory. The general idea is to first determine a relative order for each pair of offers from different services, which we can then use to form combinations of three services and then gradually include all other services until we either end up with combinations containing all services or we fail along the way.

3.3.1 Determining a relative order

The first key insight is that we can determine an optimal offer order depending on their timeframe arrangement. The arrangements shown in Figure 3.2 correspond to the following optimal order:

I. Both timeframes are equal:

$$O[x]_s = O[y]_s \land O[x]_e = O[y]_e$$

In this case the optimal order of offers can be either way as long as both can be picked in order. Being able to pick both offers is a requirement that holds true for all possible

arrangements, otherwise an offer pair can be disregarded. For TempCo the order here determined by internal `id` which each offer gets assigned on initialization.

II. Offer start times are equal but their lengths vary:

$$O[x]_s = O[y]_s \wedge (O[x]_e > O[y]_e \vee O[x]_e < O[y]_e)$$

Because one offer has a lower duration it is better to pick this offer before the longer one. The reasoning behind this is that by picking the shorter offer first we have a higher chance of picking the longer one next. The other way around we would limit the possibility space because picking the longer offer first at the earliest time possible can prevent picking the other offer more often.

III. One offer starts before the other and the end time does not exceed the other:

$$(O[x]_s < O[y]_s \wedge O[x]_e \leq O[y]_e) \vee (O[y]_s < O[x]_s \wedge O[y]_e \leq O[x]_e)$$

Similar to II. we can optimize the possibility space by always picking the earlier offer first. Note that this arrangement also includes the case in which both offers don't overlap at all, in which case it is obvious to pick the earlier offer first.

IV. In this arrangement one offer encapsulates the other:

$$(O[x]_s < O[y]_s \wedge O[y]_e < O[x]_e) \vee (O[y]_s < O[x]_s \wedge O[x]_e < O[y]_e)$$

Here both order possibilities have to be considered, just considering the earlier offer first followed by the encapsulated offer like III. might lead to no solution. We also need to account for situations like II., in which the encapsulated offer can go back into the encapsulating offer. In a way this arrangement represents a mix of cases II and III, so both possibilities have to be checked and added to the relative order.

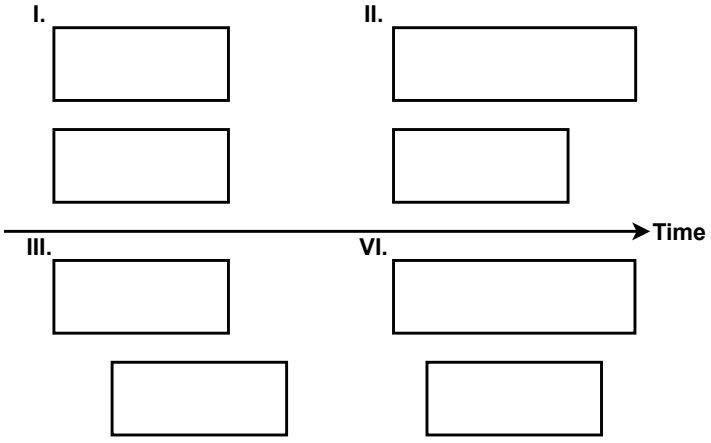


Figure 3.2: All possible two offer timeframe arrangements visualized.

The matcher stores relative ordering information in a specialized map class called `OptionMap`. For any two offers from different services we calculate and save:

1. The possible combinations itself, which are nicknamed *options* and map then with a key. We convert the pair of offers a, b to a key k and use `offerOptionMap` to map this key to the corresponding option: $k = (a, b) \mapsto [a, b]$. Whenever we need to determine any key for mapping we take the mapped objects `id` and covert them ordered to a string separated by an arbitrary symbol.
2. Append and prepend offer maps for each service/offer combination. Because the FLEXMATCH algorithm works by adding missing services to existing combinations we need a way to quickly lookup if it is possible to append or prepend a service at any given offer. This information is saved in two additional maps: `appendOfferMap` and `prependOfferMap`. The `appendOfferMap` stores keys of the form $(a, s_b) \mapsto [b, ...]$, where a, b are offers from before, s_b is the service of b and $[b, ...]$ a list of offers, likewise `prependOfferMap` works the other way around $(s_a, b) \mapsto [a, ...]$. By replacing either option key offer with the respective service, we can quickly get which offers from a service s are append- or prepend-able to any offer.
3. For the first 3 services in S we also save their corresponding options $(s_a, s_b) \mapsto [[a, b], ...]$, which is used to find combinations of length 3 in the next step.

3.3.2 Combinations of length 3

After all relative orders are determined and saved, we can use this information to find options with 3 services. If we think about combinations as permutations of the first three services $[0, 1, 2]$ then we need to explore $3! = 6$ possibilities, these are:

$$[0, 1, 2],\ [0, 2, 1],\ [1, 0, 2],\ [1, 2, 0],\ [2, 0, 1],\ [2, 1, 0]$$

We can arrive at all of these if we append or prepend the missing service to our combinations of length 2 as shown in Listing 3.6. We begin with the `basePairs` $[0, 1], [1, 0], [1, 2], [2, 1]$ (line 2 - 6) and set the `missingService` to 2 for the first two pairs and 0 for the others (9). Then we get all options associated with each pair of services stored before in step 3 (10). We then get the prepend (16) or append offers (18) for the first or second offer in each option, form a new option with the offer prepend or appended.

For pairs $[1, 0]$ and $[1, 2]$ we don't check for prepend offers because combinations $[2, 1, 0]$ and $[0, 1, 2]$ can also be formed by $[0, 1]$ & 2 and $[2, 1]$ & 0 respectively. This way it's guaranteed that no combination will have duplicate options and we don't need to keep track of which combinations were checked already.

Function `checkOfferCompatibility` in line 28 guarantees that we only add possible options to the list of `currentOptions`. We go through each offer and pick it as early as possible while keeping track of the current time. Whenever there's a gap between two offers, we set the current time

Listing 3.6 FLEXMATCH algorithm creating combinations of 3 services

```
let currentOptions: ServiceOffer[][] = [];
let basePairs = [
  [services[0], services[1]],
  [services[1], services[0]],
  [services[1], services[2]],
  [services[2], services[1]]];
for (let i = 0; i < 4; i++) {
  let servicePair = basePairs[i];
  let missingService = i < 2 ? services[2] : services[0];
  let serviceOptions = optionMap.getServicePairOptions(servicePair[0], servicePair[1]);
  serviceOptions.forEach((option) => {
    for (let prepend = 0; prepend <= 1; prepend++) {
      let nextOptions;
      if (prepend) {
        if (i == 1 || i == 2) { continue; }
        nextOptions = optionMap.getPrependOptions(missingService, option[0]);
      } else {
        nextOptions = optionMap.getAppendOptions(option[1], missingService);
      }
      if (!nextOptions) { continue; }
      nextOptions.forEach((nextOption) => {
        let newOption;
        if (prepend) {
          newOption = [nextOption, ...option];
        } else {
          newOption = [...option, nextOption];
        }
        if (this.checkOfferCompatibility(newOption)) {
          currentOptions.push(newOption);
        }
      });
    }
  });
}
```

to the next offer start time. If adding o_t to the current time exceeds the offer end time we know this option is not possible hence there's no need to consider it further. Now `currentOptions` contains all viable combinations of the first three services, if there are more we continue with the next step.

3.3.3 Extending combinations to length n

Exhausting all viable combinations of services requires an algorithm that is capable of generating all possible options that need to be checked. If we start with a list of 3 service combinations, we need to insert the next service at each position in the list, which for a list of length x is

$x + 1$ positions. We can continue using only the resulting viable combinations to add all further services until we arrive at combinations of length n. Listing 3.7 shows how this approach is implemented in TempCo.

We begin by assigning `nextServices` to all remaining $n - 3$ services which have yet to be included in the options (line 2). While there are still services left we go trough the `currentOptions` (6), trying to insert `nextService` at every possible index. Lines 8 - 15 prepend and lines 35 - 42 append the `nextService` similar to Listing 3.6, where we get the prepend or append offers associated to `nextService` and the first or last offer in the current option. If this new option is viable, we add it to the list `nextOptions` for the next iteration. Lines 16 - 33 go through each offer in the option except the last one and test if we can append any offers from `nextService` to the current offer (19). If possible, we also look to prepend the offer to the `nextOffer` in our option, which is done by checking if that offer pair exists in the `OptionMap` (24). Only then we create a `newOption` with the `nextService` offer inserted (27) and test their compatibility (28). Again only viable options are added to the list of `nextOptions` (31) which are processed in the next iteration. Also if `nextOptions` doesn't contain any options, we can conclude that there are no possible combinations with all requested services and return an error (45). However, to know which services interfere with each other exactly, we'd need to calculate the whole combination tree of each possible paring from `currentServices` and `nextService`. Which is $\sum_{k=0}^{n} \frac{n!}{k!}$ possibilities with n being the number of services and is therefore too complex to calculate within this algorithm. The only error we can reasonably return is that `currentServices` and `nextService` can't be combined without further specification.

On exit of the while loop, `currentOptions` will contain all viable options with combinations of n services. What's left is to pick concrete times for all offers and compare all options against each other. This final step is handled in the `getBestSelection` function which creates `Selection` objects that are returned as the algorithm result.

3.3.4 Defining Selections

`Selection` objects are a container for options with additional information such as picked times for each offer and total attribute values. The constructor shown in Listing 3.8 handles how attributes of a FLEXMATCH solution are initialized. Similar to the `checkOfferCompatibility` function, we begin by selecting each offer at the earliest time possible (line 6 & 8) and save all chosen start- and end-times in an array `times` (8), where all even indices represent start times while odd are end times. Due to D_{md} additional calculations are required, because of situations depicted in Figure 3.3. Here all offers were selected at the earliest time possible, which created a gap of $2D_{md}$ between o_1 and o_2. However, a solution is indeed possible by selecting o_1 later, which breaks up the wait time into two parts which are both $\leq D_{md}$ therefore making the selection valid. Lines 10 - 37 handle this moving of selected offer timeframes by iterating backwards through the defined `times` array after each offer is added. We calculate `prevDistance` with previous timeframe end to current start time (18, 30, 32) and compare it against D_{md} in

Listing 3.7 FLEXMATCH algorithm inserting the other services in the existing options

```
let currentServices = [services[0], services[1], services[2]];
let nextServices: Service[] = services.slice(3);
while (nextServices.length > 0) {
  let nextOptions: ServiceOffer[][] = [];
  let nextService = nextServices.pop()!;
  currentOptions.forEach((option) => {
    let offer = option[0];
    let prependOptions = optionMap.getPrependOptions(nextService, offer);
    prependOptions?.forEach((prependOption) => {
      let newOption = [prependOption, ...option];
      if (!this.checkOfferCompatibility(newOption)) {
        return;
      }
      nextOptions.push(newOption);
    });
    for (let i = 0; i < option.length - 1; i++) {
      offer = option[i];
      let nextOffer = option[i+1];
      let appendOptions = optionMap.getAppendOptions(offer, nextService);
      if (!appendOptions) {
        continue;
      }
      appendOptions.forEach((appendOption) => {
        if (!optionMap.hasOfferPair(appendOption, nextOffer)) {
          return;
        }
        let newOption = [...option.slice(0, i+1), appendOption, ...option.slice(i+1,
            option.length)];
        if (!this.checkOfferCompatibility(newOption)) {
          return;
        }
        nextOptions.push(newOption);
      });
    }
    offer = option[option.length-1];
    let lastOptions = optionMap.getAppendOptions(offer, nextService);
    lastOptions?.forEach((lastOptions) => {
      let newOption = [...option, lastOptions];
      if (!this.checkOfferCompatibility(newOption)) {
        return;
      }
      nextOptions.push(newOption);
    });
  });
  if (nextOptions.length == 0) {
    return new MatchingError(currentServices, nextService);
  }
  currentServices.push(nextService);
  currentOptions = nextOptions;
}
return this.getBestSelection(currentOptions);
```

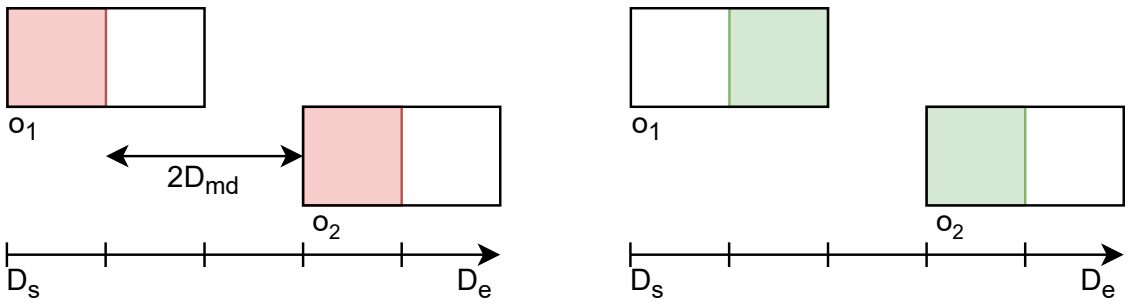


Figure 3.3: Visualization of the Selection distance problem where D_{md} is one time unit and selected offer times are coloured.

the while loop (19), which runs until we reach the beginning or a distance is smaller than D_{md}. For the previous offer `maxMoveRange` is calculated (21), that provides the maximum amount a selected offer timeframe can be moved forward. It is defined by the distance to either offer end time or next selected offer timeframe start, whichever is earlier. We then check if `maxMoveRange` is enough to satisfy D_{md} for this offer (22) and throw a `MatchingError` if it's not the case, therefore cancelling the Selection. How far an offer is moved gets calculated in line 25 where we subtract D_{md} from the previous distance. This is done to fully utilize D_{md} and maximize the available time for all previous offers. Because of the while condition `moveDistance` can't be negative and we guarantee `moveDistance` $\leq$ `maxMoveRange` with:

$$\begin{aligned} &\text{prevDistance} - \text{maxMoveRange} \leq D_{md} && \textit{*Checked in line 22} \\ \Rightarrow \quad &\text{prevDistance} - D_{md} \leq \text{maxMoveRange} \end{aligned}$$

Now `moveDistance` is added to the previous offer selected timeframe (26 & 27) and we continue to calculate `prevDistance` for each previous offer, using D_s again if we arrive at the beginning (31).

Throughout this process we sum offer cost (4), rating (5) and wait time (7) in the `attribute` property which is used for the TOPSIS method. It is possible to calculate the wait time while adding each offer, because moving a selected timeframe that is not the last element doesn't change the overall wait time but redistributes it. Each `Selection` gets ranked with TOPSIS using the demand weights provided, and we return the highest ranked `Selection` as the result. One final adjustment can be made after the best `Selection` was found: Because D_{md} is always fully utilized if a selected timeframe must be moved, situations can occur in which the wait time is not evenly distributed before and after. Doing a last forward iteration through `times` where selected timeframes are moved forward can be used to evenly distribute wait times if desired by the user.

Listing 3.8 `Selection` constructor function

```
let currentTime = demand.start;
for (let i = 0; i < this.offers.length; i++) {
  let offer = this.offers[i];
  this.attributes.cost += offer.cost;
  this.attributes.rating += offer.rating;
  let offerStart = Math.max(offer.start, currentTime);
  this.attributes.coverage += calcDistance(currentTime, offerStart);
  currentTime = offerStart + offer.serviceTime;
  this.times.push(offerStart, currentTime);
  let j = i*2;
  if (j == 0) {
    if (calcDistance(demand.start, this.times[0]) > demand.maxDelay) {
      throw new MatchingError('Demand start to first offer distance is too far!');
    } else {
      continue;
    }
  }
  let prevDistance = calcDistance(this.times[j-1], this.times[j]);
  while (prevDistance > demand.maxDelay && j > 0) {
    let prevIndex = (j - 2) / 2;
    let maxMoveRange = calcDistance(this.times[j-1], Math.min(this.offers[prevIndex].end,
        this.times[j]));
    if (prevDistance - maxMoveRange > demand.maxDelay) {
      throw new MatchingError(`Offer distance ${prevIndex} to ${prevIndex+1} is too far!`)
    }
    let moveDistance = prevDistance - demand.maxDelay;
    this.times[j-1] += moveDistance;
    this.times[j-2] += moveDistance;
    j -= 2;
    if (j > 0) {
      prevDistance = calcDistance(this.times[j-1], this.times[j])
    } else if (j == 0) {
      prevDistance = calcDistance(demand.start, this.times[0]);
      if (prevDistance > demand.maxDelay) {
        throw new MatchingError('Demand start to first offer distance is too far!');
      }
    }
  }
}
```

4 Performance & Optimizations

This chapter analyses API component performance and outlines which optimizations can be applied to increase responsiveness and throughput. Because TempCo is intended to be used in real time environments with Personal Assistants, we want to give responses as soon as possible to increase user engagement. Because any algorithm runtime gets added on top of natural language processing and network delays, optimizing these algorithms is the best way to ensure quick responses.

Initially the JavaScript library 'luxon' was used for temporal calculations but had to be replaced, because the provided convenience functions all came with a noticeable associated cost. Internally luxon works with unmutable objects that prevent any side effects but some are created in the background for various calculations. Only after profiler analysis it became apparent that this approach is not performant for often run code as it is the case with both algorithms. Adjusting all calculations to work with UNIX Epoch milliseconds provided a 5x increase in runtime and lowered RAM usage significantly for the FLEXMATCH algorithm. Restricting the luxon library for parsing date-time strings and handling output formatting proved to be much more efficient and better utilized library capabilities.

All performance tests were done with an Intel Core i5-6600K CPU clocked at 3.50GHz running Node.js v12.16.1 on Windows 10 and 8GB RAM. Each run was performed in a separate process to ensure the same starting conditions across all runs and graphs reflect the average of three runs.

4.1 FULLMATCH Algorithm Performance

The FULLMATCH and FULLMATCH simple algorithm both operate within a time complexity of $\mathcal{O}(n^2)$. This is due to the two `for`-loops where the outer loop needs to iterate through all offers while the inner iterates from the current position to list end. The worst case happens when the inner loop has to check most of the offers before reaching a break condition such as the time delay getting to large or arriving at an earliest best pick in the simple algorithm variant. To show this behaviour we split the number of offers n in two equally sized groups one temporally before the other, where each offer in the same group has equal start and end times. If we then place the groups close enough to each other, with distance less than D_{md}, we get an instance in which each offer from the earlier group has to check all offers from the later group which is $\frac{n}{2}$.

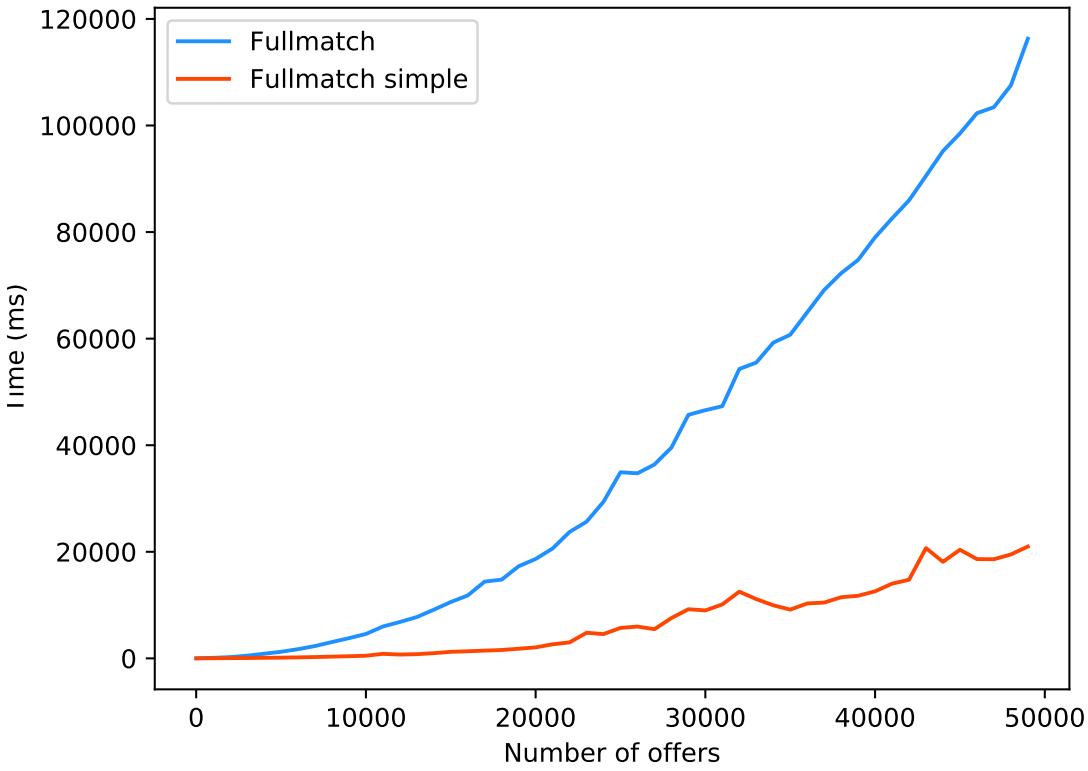


Figure 4.1: FULLMATCH algorithm comparison with $\frac{n}{2}$ options for half the offers.

The result shown in Figure 4.1 shows the quadratic growth for both algorithms. Because the simple algorithm doesn't use TOPSIS for its calculations there's a clear difference between both algorithms, which increases as more offers need to be processed. Random spikes also appear more frequently which is likely due to garbage collection requiring more time and resources.

It is unlikely that data for this algorithm is arranged in this fashion, instead offers should follow a loose schedule where there might be small fluctuations in start and end times but each offer roughly has the same choice of offers to pick from as its overlapping neighbours. If we take hotels as an example, each hotel might offer rooms with different check-in and check-out times but all rooms share overnight availability and it should be possible to switch from one hotel to another depending on D_{md}. With this in mind, Figure 4.2 shows the runtime if each offer only has $\lceil \log_2(n) \rceil$ choices for n offers, which is done through the formation of distinct overlapping offer groups that are all D_{md} apart from one another. The resulting graph now resembles a linear function for each algorithm that has a much lower runtime even with 20 times more offers as shown in 4.1. Where before only up to ten thousand offers could be processed in a reasonable amount of time, now up to a million offers can be processed in a

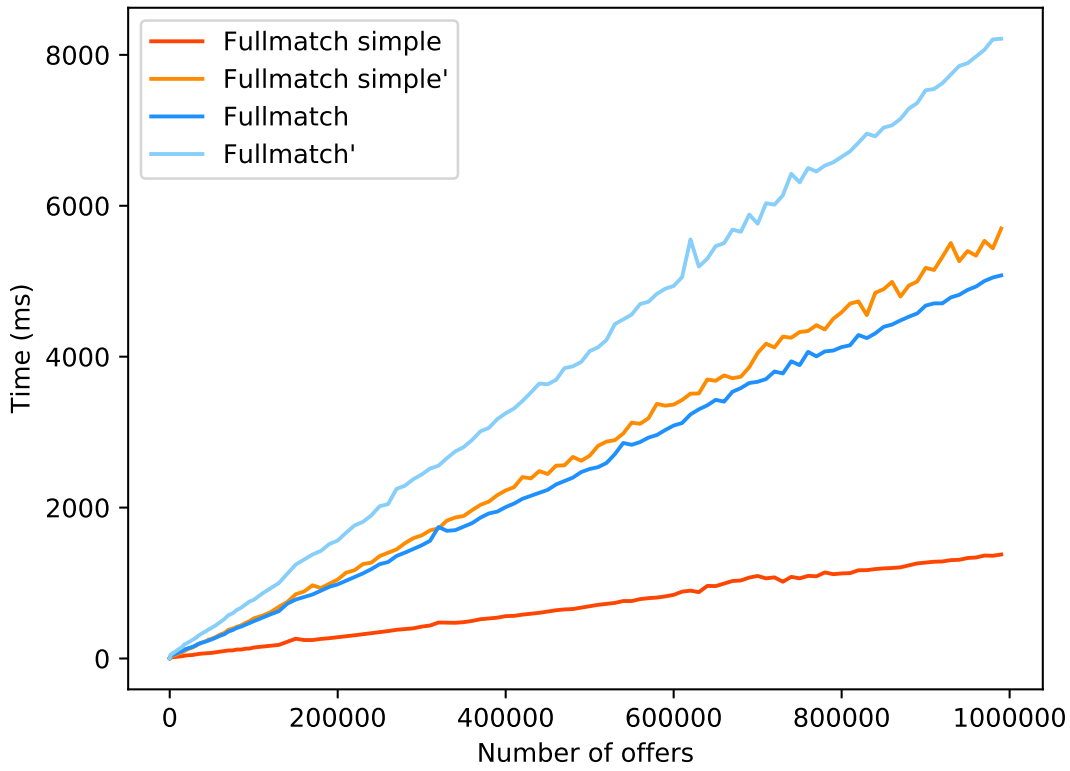


Figure 4.2: FULLMATCH algorithm comparison with $\lceil \log_2(n) \rceil$ choices per offer.

fraction of the time. At these high sample sizes even small optimizations can cause a huge difference as graphs marked with a tick show the performance before any optimization. After changing the way total attribute values were stored and calculated for each offer and by only storing the best result within the TOPSIS calculation, a performance boost for the FULLMATCH algorithm was achieved that even surpassed the old simple algorithm. Up until 200,000 offers both algorithms can now finish within a second, which is quite fast considering that there are $\lceil \log_2(200,000) \rceil = 18$ options for each offer with over 11 thousand distinct offer groupings. The simple variant stays below one and a half seconds well after one million offers.

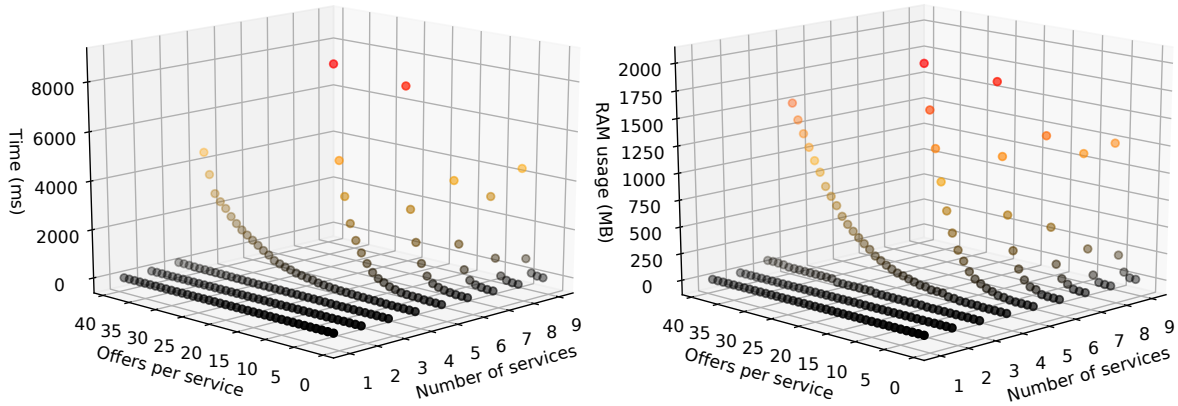
With increase in available choices the resulting runtime will more closely resemble $\mathcal{O}(n^2)$ but it can get as low as $\mathcal{O}(n)$ if the conditions are right. In practice offers will not consistently follow any predictable pattern, so the true runtime will most likely be in between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ which is entirely dependent on the input.

4.2 FLEXMATCH Algorithm Performance

Because the FLEXMATCH problem is much more difficult to solve exactly, we can't easily achieve the same reasonable results as FULLMATCH. As Figure 4.3a shows, we begin to hit hard resource limits as the number of services increases. When inspecting the worst case scenario: Each service offer is compatible with all other service offers which causes the number of combinations to increase exponentially. At four services the runtime begins to show exponential growth and increases with more services. This also reflects in the RAM usage which also grows exponentially shown in Figure 4.3b. After 5 services with 20 offers each, the Node.js process crashes with the fatal error: 'Allocation failed - JavaScript heap out of memory'. Because the algorithm has to step through each possible combination after length 3 and inserts a new service, the `nextOptions` increases exponentially as each offer is compatible at all positions. This consumes more and more heap memory until we cannot allocate any more and the process must be terminated. We get the expected worst case runtime roughly by multiplying all services offer counts:

$$\prod_{s \in S} |s_o| \approx \mathcal{O}(n^{|S|}) \tag{4.1}$$

Services with just one offer can easily be processed as shown in the performance graphs but just one service with a lot of offers can drastically increase runtime and RAM usage. However, when considering a more constrained arrangement where offers are not easily compatible performance can increase: Because we check for each new combination if it is compatible, there's always the possibility of a decrease in options for the next iteration. Depending on which services are included and the resulting options, there exists an optimal order that minimizes the total amount of processed options. Finding this option would be key in decreasing average



(a) Runtime **(b)** RAM usage

Figure 4.3: FLEXMATCH algorithm worst case performance graphs in relation to offer and services count.

runtime and RAM usage but is out of scope for this thesis. Without optimization, practically only up to 3 services can be matched without strict limitations, otherwise equation 4.1 has to be considered together with available resources if the algorithm is able to provide an answer.

4.3 Parsing Performance

One key component in TempCo is the Parser, only after all data has been parsed the algorithm can begin execution, which means any time spent parsing is directly added to the total runtime. Testing both parsing methods with the simplest possible form of offer data structure, provided the results depicted in Figure 4.4. Clearly both graphs are in $\mathcal{O}(n)$ but the results are not fit for real time use. Compared to the performance of FULLMATCH in Figure 4.2 its clear that parsing one million offers will be much slower than actually running the algorithm. Doing a simple linear regression shows, that parsing one million offers would take about 3 minutes and 20

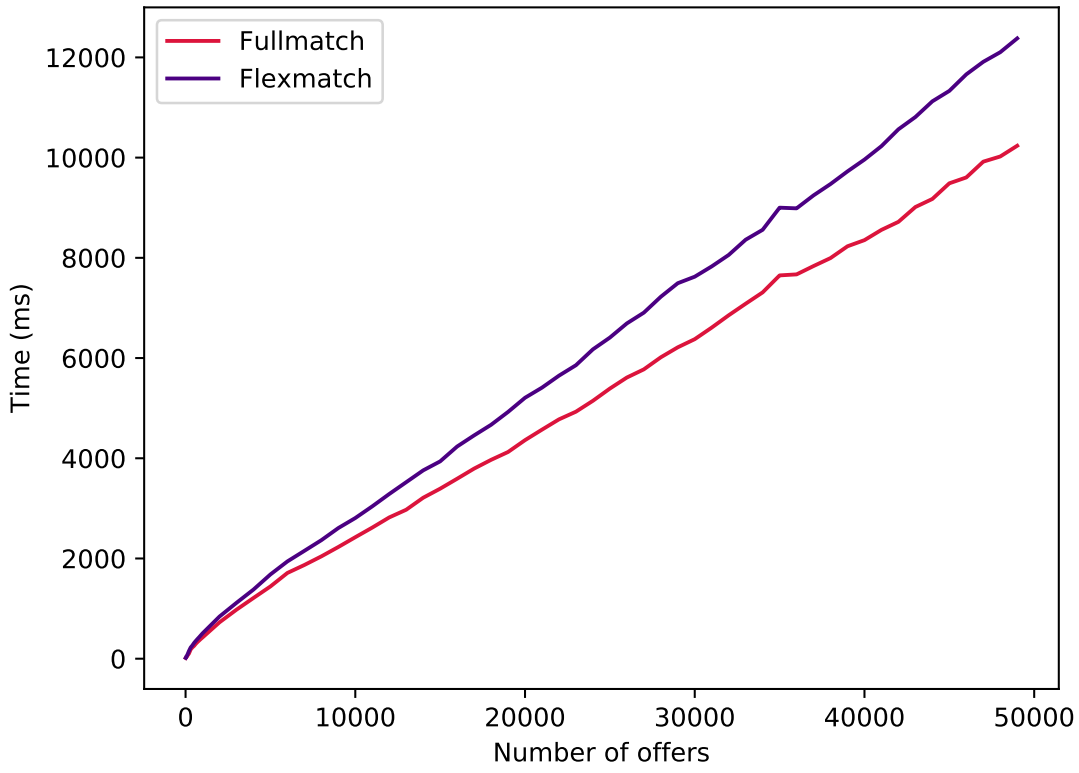


Figure 4.4: Comparison of parsing offers for both algorithms using simple offer objects.

seconds which is not feasible in any real time environment. Unfortunately using the profiler reveals, that most time spent while parsing is within the JSON-LD library used for framing the offer data. Framing a JSON-LD structure turns out to be a very costly operation where each object first has to be expanded, framed and then compacted again for the output. Internally this requires a lot of additional memory that gets freed later by the garbage collector which slows everything down even further. Therefore, RAM usage also varies depending on when the last Mark-sweep for garbage collection happened.

5 Use Cases

This chapter presents three possible usage scenarios for TempCo and shows how the API can be integrated in different domains. All scenarios presented here are focused on Personal Assistant usage; however, as described earlier, TempCo is not limited to any specific domain and can be adapted to a variety of uses. Depending on the capabilities of each machine the algorithms are either executed locally or in the cloud.

5.1 Smart Home

In a smart home, the user has a vacuum cleaner that works autonomously and a house layout with multiple rooms that should be cleaned regularly. The vacuum cleaner knows how long it needs to clean each room without interruption; however, the user has invited guests that occupy certain rooms and would make cleaning difficult. Because the user still wants all rooms cleaned and maybe clean-up after their guests, the FLEXMATCH algorithm could calculate the quickest order to clean all rooms. Each room would be modelled as a service and the times when a room can be cleaned become offers for the vacuum cleaner. With knowledge about how long each room takes to clean the offer timeframes can be set accordingly and after certain time points e.g. after the guests leave. FLEXMATCH would help make home automation even smarter.

5.2 Booking and Reservations

Here we have a user that wants to visit a city and stay in hotels for a given duration. Each Hotel offers different room types with set check-in and check-out times. Because the user is booking last-minute, no hotel can provide a single room for the full duration of the stay, but if booking multiple rooms across hotels the full duration could be covered. By using the FULLMATCH algorithm we can find the best combination of hotel rooms to cover the intended stay. Every hotel in the city area would need to provide for each night which room types are available, the respective check-in/out times, price and a review score that could be provided through a review aggregation site. Each overnight stay in a room is modelled as an offer, with which the FULLMATCH algorithm calculates the cheapest combination, the best rated one or something that balances all aspects based on user preference weights. Because offers with the

same resource i.e. same hotel room, their stays are combined if they are on adjacent nights, giving them an implicit coverage bonus that can be lessened through the demand coverage weight. Too many switches between hotels are therefore implicitly discouraged unless explicitly requested.

5.3 City Tours

This scenario depicts a user that wants to visit certain venues in a city e.g. a famous museum, a clothing store and eat their favourite dish in a restaurant. Here FLEXMATCH can be used to derive a tour on how to visit each venue during their opening hours even with little time available. All venues are represented as a service with offers representing alternatives e.g. pizza can be ordered in different restaurants while a clothing store can have multiple subsidiaries. This way different offerings can be represented and compared against each other when choosing an optimal order. Cost and ratings are taken into account through demand weights which reflect user preferences. Services like Google Maps, Yelp or Foursquare can be queried to obtain all relevant data and run the FLEXMATCH algorithm; however, the service time required to utilize an offer would need to be approximated or guessed if there's no data available. As long as all venues are in relative proximity this could provide more utility and planning capabilities to Personal Assistants.

6 Conclustion and Outlook

This Bachelor Thesis presented algorithms to solve the problems FULLMATCH and FLEXMATCH, which coordinate services and offers temporally. Furthermore, the developed TempCo API leverages the semantic web through JSON-LD and can parse data from various sources by using the Schema.org vocabulary. The algorithms were implemented in TypeScript which, when compiled to JavaScript, allows the code to be portable and run on different platforms. A standalone Node.js HTTP REST server implementation is also provided that listens to post requests enabling algorithm usage through the network. Performance of all algorithms were analysed in chapter 4 to identify practical limits and runtime complexities. Results show that parsing JSON-LD objects can be slower than running the FULLMATCH algorithms and the FLEXMATCH algorithm can only be used for small or tightly constrained datasets. TOPSIS is utilized to rank alternatives based on user preference whenever there are multiple options available with attribute trade-off potential. Possible use cases were detailed in chapter 5 showing the versatility and applications of both algorithms.

Outlook

While the developed API already has some potential uses cases, it still needs to be refined more before any commercial use. The FLEXMATCH algorithm could be much more time and memory efficient if it utilized an heuristic approach that respects the imposed real time limitations and works on larger datasets. Combining Constraint Programming with Large Neighbourhood Search as presented by Hentenryck [Hen16] shows good results for the related Asymmetric Travelling Salesman Problem with Service Time and might be able to use parts of the presented FLEXMATCH algorithm. Network delays need to be considered and dynamic data sources are required for mainstream use of TempCo in Personal Assistants. To further improve API capabilities other modules will need to be integrated with TempCo like a spatial component that provides data regarding travel times and reachability. Right now TempCo is limited to coordinating services in proximity or where travel times are not an issue, which is limiting in a tightly interconnected world. The semantic web with JSON-LD has proven to be a useful concept when trying to extract data from multiple different sources but the technology still requires mainstream adoption. Vocabularies like Schema.org are a step in the right direction; however, modelling every application according to a universal vocabulary is probably impossible due to different requirements and minute adjustments. While it is

always possible to define your own schemas, these are tied to your application and won't be of much help when trying to extract specific data without some form of translation. Furthermore, collecting and providing data has become a business model that is only worth as much as the amount of data available. Companies sell the rights to access their datasets, monitor any interaction and restrict how their data can be used. Even if every service could publish all semantic data through their web pages, aggregation sites and search machines are still required to find this data. While there has been a push for Linked Data as Google and other search providers are promoting rich results that show at a glance what content a site has to offer using JSON-LD, the adoption rate is still quite low based on Figure 6.1.

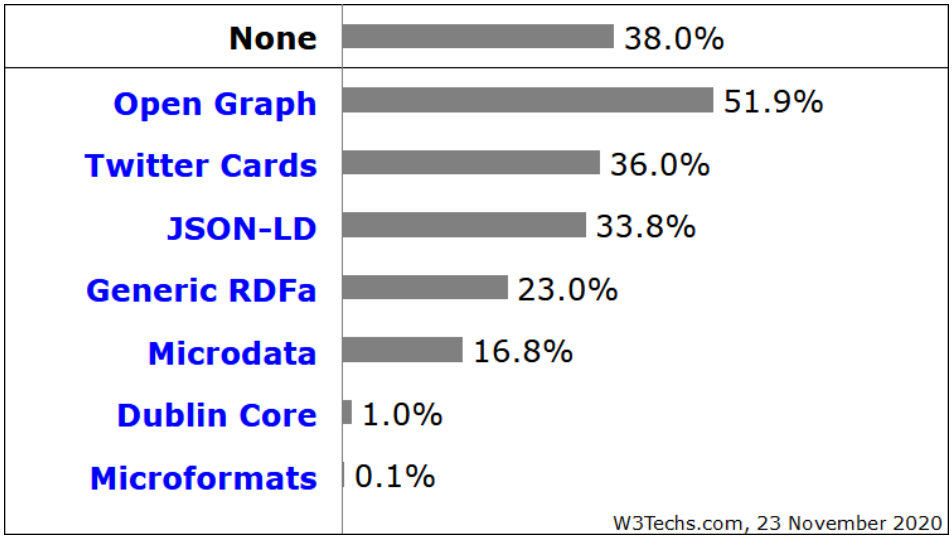


Figure 6.1: Usage statistics of structured data formats for the top 10 million websites according to Alexa and Tranco ranking. A website may use more than one structured data format [W3T20]

JSON-LD is the highest used general purpose structured data format but social media objects dominate this statistic with Twitter Cards and Facebook's Open Graph. The available data on each website can also be vastly different, offering no guarantees for availability. As JSON is the de facto standard for most web APIs it would be much easier converting to JSON-LD than a completely different format. Depending on how the Web evolves, applications like TempCo must be updated or replaced by better methods. For now, TempCo is a small step in order to make tomorrows Personal Assistants smarter and more versatile.

A Zusammenfassung

Personal Assistants wie Alexa, Cortana oder Siri haben viele Anwendungsmöglichkeiten jedoch benötigen einige spezielle APIs um gestellte Probleme zu lösen. Diese Bachelorarbeit definiert die Probleme FULLMATCH und FLEXMATCH formal, welche eine Koordinierung von Services auf der zeitlichen Ebene verlangen. Die entwickelte API, genannt ”TempCo”, löst diese Probleme und ihre Komponenten werden hier beschrieben sowie im Detail erklärt. Über JSON-LD wird das Semantic Web ausgenutzt um Daten aus verschiedenen Quellen durch Nutzung des Vokabulars von Schema.org zu parsen. Alle Algorithmen wurden in TypeScript geschrieben, wodurch der Code portabel ist und auf unterschiedlichen Plattformen laufen kann. Eine eigenständige Node.js HTTP REST Server Implementation steht auch zur Verfügung, die durch POST Requests eine Nutzung der Algorithmen im Netzwerk erlaubt. Wir analysieren die Algorithmen auf Performanz, um praktikable Limits und Laufzeitkomplexitäten zu identifizieren. Die Resultate zeigen, dass das Parsen von JSON-LD Objekten sehr langsam für Echtzeitumgebungen sein kann und dass der FLEXMATCH Algorithmus am besten für kleine oder eingeschränkte Eingaben funktioniert. Der FULLMATCH Algorithmus dagegen skaliert sehr gut und kann typische Probleminstanzen innerhalb von Sekunden lösen. Abschließend beschreiben wir mögliche Anwendungsfälle, die die Vielseitigkeit und Einsatzgebiete beider Algorithmen zeigt um Personal Assistants zu verbessern.

Bibliography

[AAB+15] R. Agerri, X. Artola, Z. Beloki, G. Rigau, A. Soroa. Big data for Natural Language Processing: A streaming approach. *Knowledge-Based Systems*, 79:36–42, 2015. (Cited on page 9)

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. (Cited on page 11)

[ANM16] H. Al Najada, I. Mahgoub. Autonomous vehicles safe-optimal trajectory selection based on big data analysis and predefined user preferences. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 1–6. IEEE, 2016. (Cited on page 9)

[AS87] E. M. Arkin, E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987. doi:10.1016/0166-218X(87)90037-0. (Cited on page 10)

[AV15] Y. Azar, A. Vardi. TSP with Time Windows and Service Time. *arXiv:1501.06158 [cs]*, 2015. URL http://arxiv.org/abs/1501.06158. ArXiv: 1501.06158. (Cited on page 11)

[BDR20] A. Bretin, G. Desaulniers, L.-M. Rousseau. The traveling salesman problem with time windows in postal services. *Journal of the Operational Research Society*, 0(0):1–15, 2020. doi:10.1080/01605682.2019.1678403. URL https://doi.org/10.1080/01605682.2019.1678403. (Cited on page 11)

[DDGS95] Y. Dumas, J. Desrosiers, E. Gelinas, M. M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995. (Cited on page 11)

[EFMS05] K. Elbassioni, A. V. Fishkin, N. H. Mustafa, R. Sitters. Approximation Algorithms for Euclidean Group TSP. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, M. Yung, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pp. 1115–1126. Springer, Berlin, Heidelberg, 2005. doi:10.1007/11523468_90. (Cited on page 11)

[FLM02] F. Focacci, A. Lodi, M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002. (Cited on page 11)

[Gar20] D. Gartzman. A Dynamic Programming Approach to Set-TSP, 2020. URL https://towardsdatascience.com/set-tsp-because-there-is-more-than-one-place-to-get-bread-712fdb5b381. (Cited on page 11)

[Hen16] P. V. Hentenryck. Large Neighborhood Search - asymmetric TSP with time windows, 2016. URL https://www.coursera.org/lecture/discrete-optimization/large-neighborhood-search-asymmetric-tsp-with-time-windows-DUgp4. (Cited on page 47)

[HY81] C.-L. Hwang, K. Yoon. Methods for Multiple Attribute Decision Making, 1981. doi:10.1007/978-3-642-48318-9_3. (Cited on page 27)

[Joy20] Joyent. Node.js website, 2020. URL https://nodejs.dev/. (Cited on page 13)

[KNC07] M. Y. Kovalyov, C. T. Ng, T. C. E. Cheng. Fixed interval scheduling: Models, applications, computational complexity and algorithms. *European Journal of Operational Research*, 178(2):331–342, 2007. doi:10.1016/j.ejor.2006.01.049. URL http://www.sciencedirect.com/science/article/pii/S0377221706003559. (Cited on page 10)

[Mic20] Microsoft. TypeScript website, 2020. URL https://www.typescriptlang.org/. (Cited on page 14)

[OT07] J. W. Ohlmann, B. W. Thomas. A Compressed-Annealing Heuristic for the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, 19(1):80–90, 2007. doi:10.1287/ijoc.1050.0145. URL https://pubsonline.informs.org/doi/abs/10.1287/ijoc.1050.0145. (Cited on page 11)

[sch20] Schema.org website, 2020. URL https://schema.org/docs/about.html. (Cited on page 18)

[Sid20] Sideway. hapi framework website, 2020. URL https://hapi.dev/. (Cited on page 14)

[Sla97] P. Slavík. The errand scheduling problem. *Computer Science Technical Report*, pp. 97–2, 1997. (Cited on page 11)

[SU10] R. F. da Silva, S. Urrutia. A General VNS heuristic for the traveling salesman problem with time windows. *Discrete Optimization*, 7(4):203–211, 2010. doi:10.1016/j.disopt.2010.04.002. URL http://www.sciencedirect.com/science/article/pii/S1572528610000289. (Cited on page 11)

[SU17] R. F. da Silva, S. Urrutia. The Traveling Salesman Problem with Time Windows (TSPTW) - Approaches & Additional Resources, 2017. URL https://homepages.dcc.ufmg.br/~rfsilva/tsptw/. (Cited on page 11)

[TH11] G.-H. Tzeng, J.-J. Huang. *Multiple attribute decision making: Methods and applications*. 2011. (Cited on page 27)

[UNE20] UNECE. Code List Recommendations, 2020. URL https://www.unece.org/uncefact/codelistrecs.html. (Cited on page 20)

[W3T20] W3Techs. Usage statistics of structured data formats for websites, 2020. URL https://w3techs.com/technologies/overview/structured_data. (Cited on pages 6 and 48)

[WBH20] M. de Weerdt, R. Baart, L. He. Single-machine scheduling with release times, deadlines, setup times, and rejection. *European Journal of Operational Research*, 2020. doi:10.1016/j.ejor.2020.09.042. URL http://www.sciencedirect.com/science/article/pii/S0377221720308468. (Cited on page 10)

All links were last followed on November 23, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature