Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Arvy Heuristics for Distributed Mutual Exclusion

Bachelor's Thesis

Silvan Mosberger

msilvan@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Pankaj Khanchandani, András Papp
Prof. Dr. Roger Wattenhofer

November 20, 2019

Acknowledgements

I thank my advisors Pankaj and András for the weekly discussions, which helped greatly in guiding the direction of this work and developing new ideas. They helped me focus on the important bits without getting distracted too much. This is in contrast to all the cats wanting to sit on my lap or desk all the time, which I'm still grateful for however. I also thank my colleagues for giving me company when I needed it and being good friends.

Abstract

In this thesis we look at different heuristics for the Arvy-class of algorithms for solving the distributed mutual exclusion problem, focusing on sequential requests. This includes the previously known algorithms Arrow and Ivy but many new ones as well. We will see that Arrow with a star tree is one of the best solutions in many cases. However we also show that a new heuristic specialized on nested clique-like graphs can still have better performance. In addition another new heuristic based on measuring request probabilities allows for better performance with adversarial requests. To evaluate the running time of Arvy heuristics, a flexible Haskell library was developed whose types guarantee correctness at compile time.

Keywords: Arrow, Ivy, distributed directory protocol, distributed mutual exclusion

CR Categories: Network algorithms

Contents

Chapter 1

Introduction

In many areas of computing there are cases where a single resource has to be shared between multiple components, with only one of them allowed to access it at a time. Components can request the resource at any time, after which they should receive it eventually. In a distributed setting an algorithm to solve this is called a distributed mutual exclusion algorithm or distributed directory protocol. An easy solution is a home-base algorithm, where a single node is dedicated to handling all requests sequentially, with the disadvantage of a lot of traffic and contention for many concurrent requests. Better solutions to this problem include the spanning-tree-based Arrow (also known as Raymond's algorithm) and Ivy protocols. Both of these algorithms can be generalized to the Arvy [1] algorithm, which allows for a wide range of different heuristics to be used for determining its behavior, while still guaranteeing correctness. There also exist other kinds of algorithms to solve this problem based on message broadcasting or quorums.

In this thesis we come up with new Arvy heuristics and measure their empirical performance in a simulation of sequential requests.

1.1 Related Work

Raymond [2] originally introduced an Arrow algorithm where a radiating star tree was found to be the best topology. Simulations showed that randomly constructed trees have an average diameter of about $\log n$ which was also found to be a good approximation for the number of messages necessary for a request. Under heavy load however it was shown to only require about 4 messages. Also for random trees, Rényi and Szekeres [3] showed the expected tree height from an arbitrary root node to be about $\sqrt{2n\pi}$. Later Demmer and Herlihy [4] independently developed the same Arrow algorithm. They analyzed its complexity for a minimum spanning tree and found it to be optimal within a factor of $\frac{1}{2}(1+\text{MST-stretch}(G))$. Kuhn and Wattenhofer [5] proved that Arrow is $\mathcal{O}(\log D)$-competitive, where D is the diameter of the spanning tree it operates on. Herlihy and Warres [6] compared Arrow to a home-base algorithm and found Arrow to perform much better under load. Ghodselahi and Kuhn [7] showed that the competitive ratio of the

Arrow protocol is constant on certain tree topologies. Peleg and Reshef [8] showed that Arrow on the best shortest-path tree has at most 2 times longer paths than in the graph.

Ivy was introduced before Arrow by Li and Hudak [9]. An amortized bound of $\mathcal{O}(\log n)$ for path reversal was shown by Ginat et al. [10] which is equivalently Ivy's number of messages per request. In [11] this bound was shown to be tight with a tree that requires $\log n$ many messages for every request in a specific request sequence.

Arvy was newly introduced by Khanchandani and Wattenhofer [1] where it was shown to be correct in asynchronous networks with concurrent requests in arbitrary topologies. Since Arvy generalizes both Arrow and Ivy, this proves their correctness as well. In addition they show an Arvy heuristic with constant competitive ratio on ring graphs.

1.2 Model

We consider a complete graph $G = (V, E)$ with n vertices and $\binom{n}{2}$ edges.

Metric Costs Each edge has a cost/distance $c : E \to \mathbb{R}^+$ associated with it, representing the amount of time it takes for a message to traverse it. This function forms a metric space:

- The cost from a node to itself is zero: $\forall v : c(v, v) = 0$
- Costs between different nodes are positive: $\forall u, v : u \neq v \Rightarrow c(u, v) > 0$
- Costs are the same in both directions: $\forall u, v : c(u, v) = c(v, u)$
- The triangle inequality holds: $\forall u, v, w : c(u, w) \leq c(u, v) + c(v, w)$

As a reasonable constraint, a node can only query costs for edges to its neighbors. In a realistic scenario it's these costs that can be obtained and updated by doing regular pings to other nodes. While we don't consider changing costs in our model, there is no inherent reason the heuristics we describe can't work in such cases.

Instantaneous Computations In addition, every node is modeled as a machine with the ability to execute arbitrary effectful code such as reading/writing state or generating randomness. We model execution to be instantaneous such that no time passes during computations. Therefore traversing graph edges are the only place to spend time on.

Sequential Requests A major simplification we make is that nodes can only request the token sequentially. So only after the token has reached the requesting node, another node can issue another request. Therefore requests can be represented as a series $R = \{r_1, r_2, \dots\}, r_i \in V$ where r_i encodes the i-th request originating from node r_i. While this series could be infinite, we let it be finite such that we can reason about its total running time.

Request Paths In the type of algorithms we will look at, when the i-th request is made by node r_i, the request travels along some path $A^{(i)} = \{a_0^{(i)}, a_1^{(i)}, \dots, a_{l_i}^{(i)}\}$ with $l_i = |A^{(i)}| - 1$ being the index of the last node and also the number of messages that are being sent. The first node in this path $a_0^{(i)} = r_i$ is the node issuing the request, while the last one is currently holding the token. After the request was sent along this path, the token is then sent directly from $a_{l_i}^{(i)}$ to $a_0^{(i)}$ to fulfill the request. If the requesting node has the token already, then $|A^{(i)}| = 1$. It holds that $a_0^{(i)} = a_{l_{i+1}}^{(i+1)}$ for all $i \in \mathbb{N}$, meaning the node making a request in one request will be the last node in the request path of the following request.

Performance In order to be able to talk about performance of such algorithms, we first define $c(A^{(i)})$ to be the cost of traversing request path $A^{(i)}$, which will be the sum of the costs of the edges it includes:

$$c(A^{(i)}) \coloneqq \sum_{k=1}^{l_i} c(a_{k-1}^{(i)}, a_k^{(i)}) \tag{1.1}$$

For graph cost normalization we also define c_{avg} to be the average cost of an edge in the graph:

$$c_{avg} = \frac{1}{|E|} \sum_{(u,v)\in E} c(u, v) \tag{1.2}$$

With this in place we define performance as follows

- Arguably the most important performance measure for comparing different algorithms is the time it takes to satisfy the requests. For a single request this includes the cost to traverse the request path as well as the cost to send the token back to the requesting node. However since the cost to send the token back does not change between different algorithms, we ignore it to get a simpler measure for comparison. To normalize the costs, we divide by c_{avg}, and to get a result over all requests R we take the average over all individual requests. With this we get the definition for the average request time $\mathcal{C}_{time}$:

$$\mathcal{C}_{time}(R) = \frac{1}{|R|} \sum_{i=1}^{|R|} \frac{c(A^{(i)})}{c_{avg}} \tag{1.3}$$

- Another interesting measure is the number of messages being sent per request, also known as the hop count. While this isn't very useful with the restrictions of instantaneous execution and sequential requests in our model, it can give us a sense of how well algorithms would do without them. We use the following definition for the average request hops $\mathcal{C}_{hops}$:

$$\mathcal{C}_{hops}(R) = \frac{1}{|R|} \sum_{i=1}^{|R|} l_i \tag{1.4}$$

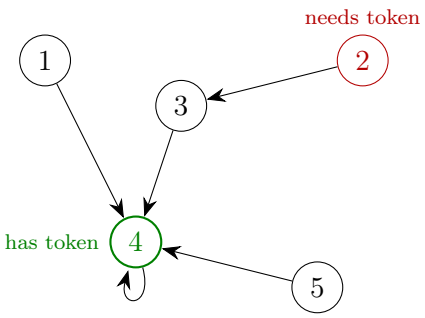
1.3 How Arvy Works

In this section we describe the Arrow and Ivy algorithms in detail, followed by the generalization to the Arvy-class of algorithms.

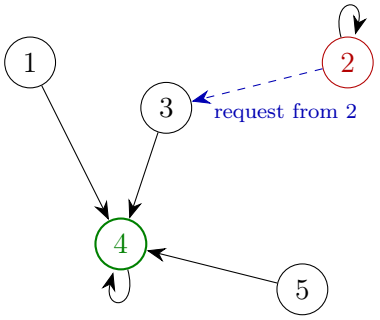
All of these are based on the idea of maintaining a rooted spanning tree over time: Every node stores a pointer $parent : V \to V$ pointing to its parent in the tree, while the root node points to itself. When a (non-root) node a_0 needs the token, it sends a request message towards its parent $a_1 = parent(a_0)$. When node a_i receives such a request, it forwards it to its own parent $a_{i+1} = parent(a_i)$ and so on until the root a_l containing the token is reached. This forms a request path $A = \{a_0, a_1, \ldots, a_l\}$. The final node then finishes its own work with the token after which it gets sent directly to a_0 such that the request is fulfilled. Now to make this a functioning algorithm, the parent pointers along this request path need to be inverted in some way to restore the rooted tree. Such a heuristic for inverting the pointers is what differentiates Arvy algorithms from each other. We will now look at what heuristics Arrow, Ivy and Arvy in general use.

1.3.1 Arrow

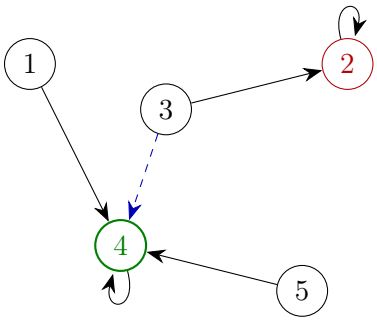
The Arrow algorithm is the simplest way to maintain a rooted spanning tree, in that it just inverts the pointers along the request path: When a_{i+1} receives a request from a_i, it sets $parent(a_{i+1}) = a_i$. See Figure 1.1 for a walk-through of a request with Arrow. Notable with Arrow is that it doesn't ever change the structure of the spanning tree, meaning that the initial tree is crucial for its performance.



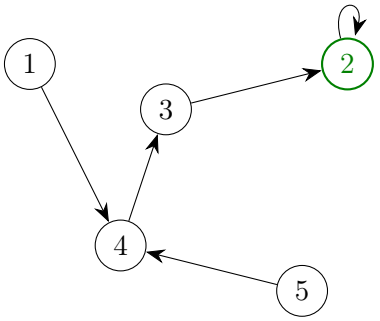
(a) Node 2 needs the token which is currently at node 4, which is the root of the spanning tree



(b) Node 2 sends a request towards its parent node 3 after which it sets its parent to itself. Temporarily there are multiple root nodes



(c) Node 3 receives the request and forwards it to its parent node 4 after which it sets its parent to node 2 from which it received the request

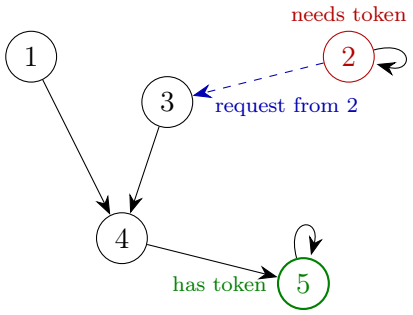


(d) Node 4 receives the request, resulting in it sending the token back to the requesting node 2. Meanwhile it updates its parent to node 3 from which it received the request from. The rooted spanning tree is now restored

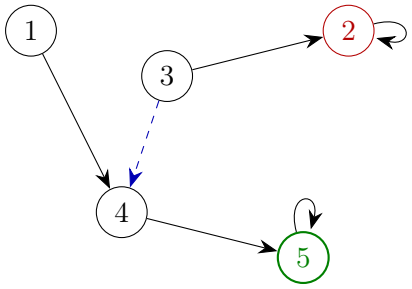
Figure 1.1: Arrow example

1.3.2 Ivy

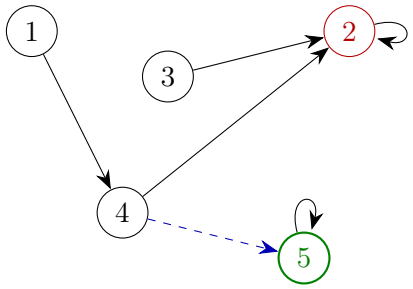
Ivy uses a different strategy for inverting the arrows, namely that every node a_{i+1} receiving the request sets its new parent to the node that made the request: $parent(a_{i+1}) = a_0$. Therefore a_0 ends up being the center of a star consisting of all nodes along the request path. Intuitively this should give good performance since shortcuts to the new root node get created, which should allow future requests to find the token faster. Figure 1.2 shows an example of Ivy.



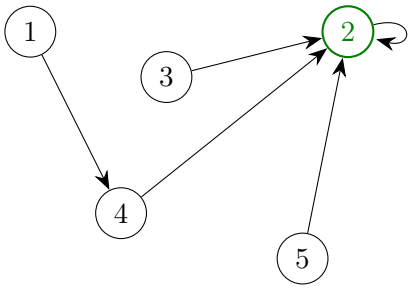
(a) Node 2 needs the token which node 5 currently has. It sends a request message towards its parent node 3 after which it sets its parent to itself



(b) Node 3 forwards the request to node 4, then it sets its parent to node 2 since that's the node that sent the original request



(c) Node 4 receives the request and forwards it further, then it sets its parent to node 2 as well since it's is the requesting node

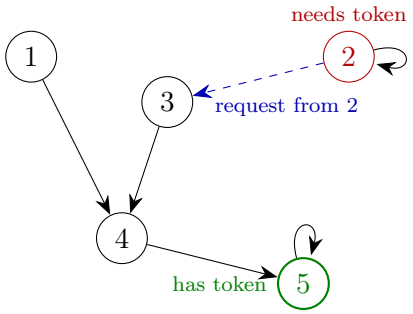


(d) The request arrives at node 5 which is holding the token. It sends the token to the requesting node 2, then it sets the parent to node 2 as well. A star including all nodes in the request path is formed, centered around the requesting node

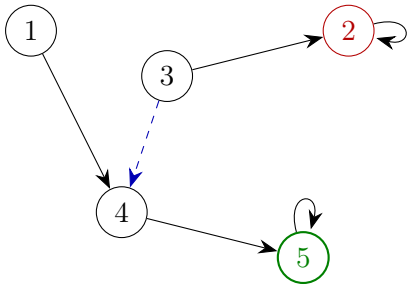
Figure 1.2: Ivy example

1.3.3 General Arvy

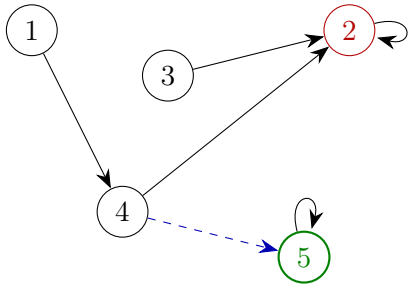
Arvy is a class of algorithms generalizing Arrow and Ivy by allowing nodes a_{i+1} that received a request to set their parent to any node the request already traveled through, so $parent(a_{i+1}) \in \{a_0, \ldots, a_i\}$. In the case that the requesting node a_0 is selected, this is equivalent to Ivy. If the node from which the request was received from a_i is chosen, this is equivalent to Arrow. Arvy therefore generalizes both Arrow and Ivy. See Figure 1.3 for an example where each node chooses a random new parent out of the available ones.



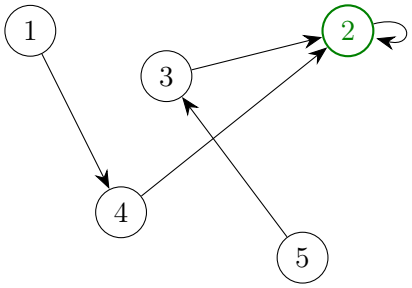
(a) Node 2 needs the token so it sends a request for it towards its parent node 3



(b) Node 3 receives the request, forwards it and chooses the only possible new parent node 2



(c) Node 4 receives the request and forwards it. It can now choose to connect back to either node 2 or 3, since the request has traveled through both of them. It chooses 2



(d) Node 5 receives the request for the token, making it send it to the requesting node 2. For selecting its new parent, it can choose between 2, 3 and 4, since all of those nodes have been traveled through already, it chooses node 3

Figure 1.3: Arvy example

Chapter 2

Algorithms

This chapter describes all the Arvy heuristics to be tested. A simplified version of a general algorithm can be written as follows. Here we use $A_k = \{a_0, a_1, \ldots, a_k\}$ to denote the nodes on the request path already traveled through, while a_{k+1} is the node that needs to select a new parent out of A_k.

Algorithm 1 Arvy algorithm

```
function RequestToken(a_0)                          ▷ Node a_0 wants the token
    if parent(a_0) ≠ a_0 then
        send request for token to parent(a_0)
        parent(a_0) ← a_0
    end if
end function
function ReceiveRequest(a_{k+1})                    ▷ Node a_{k+1} receives a request
    if parent(a_{k+1}) = a_{k+1} then
        send token to a_0
    else
        forward request to parent(a_{k+1})
    end if
    parent(a_{k+1}) ← SelectNewParent(A_k)
end function
```

All Arvy heuristics will have to define an implementation of SelectNewParent.

2.1 Arrow

As already explained in Subsection 1.3.1, Arrow maintains the structure of the spanning tree by always selecting the most recent node as the new parent, therefore inverting the arrows.

function SelectNewParent(A_k)
  return a_k
end function

2.2 Ivy

Also explained already in Subsection 1.3.2, Ivy always connects every node along a request path to the node that made the original request

function SelectNewParent(A_k)
  return a_0
end function

2.3 Uniformly Random

Not particularly interesting, but good as a reference point is a completely random heuristic, selecting a new parent uniformly random from all available choices.

function SelectNewParent(A_k)
  $a \leftarrow_{u.a.r.} A_k$
  return a
end function

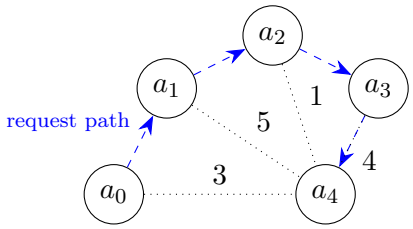
2.4 Edge Cost Minimizer

Since algorithms have access to the edge costs, we should make use of them. A simple heuristic using them is one that always chooses the node with minimum edge cost as the new parent, see Figure 2.1 for an example. Intuitively, this heuristic should perform well since short paths make for short request paths.

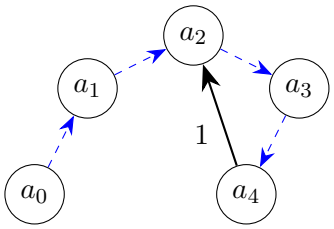
function SelectNewParent(A_k)
  return a_i such that $c(a_{k+1}, a_i), i \in \{0, \ldots, k\}$ is minimal,
   use highest i for tie-breaking
end function

Notable properties of this algorithm include

- The total edge cost in the tree can never get bigger, since a previous edge between a_k and a_{k+1} is also a candidate in the parent selection.
- This in turn means that with a good enough distribution of nodes initiating requests, the tree will eventually converge to a minimum spanning tree with which total edge cost is lowest



(a) The request arrives at a_4 which now needs to decide which one of $a_0, \ldots, a_3$ to choose as a new parent. For this it can access the edge costs to all of them.



(b) a_2 gets chosen as the new parent since the edge towards it has the smallest cost.

Figure 2.1: Edge Cost Minimizer example with four node in the request path.

- Consequently, once the tree has converged, this algorithm behaves exactly the same as Arrow, which never changes the tree by design. Therefore to get the asymptotic behavior of this heuristic, one can simply start with a minimum spanning tree directly and run Arrow on it. This seems to imply that in general this heuristic isn't any more powerful than Arrow.

2.5 Local Pair Distance Minimizer

An even better measure to minimize is the total distance between all pairs of nodes in the spanning tree T, defined as

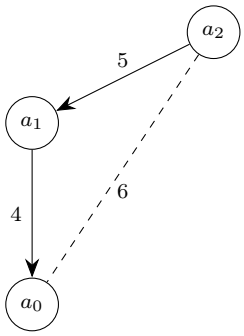
$$\mathcal{C}_{pairs}(T) = \sum_{u,v \in V, u<v} d_T(u,v) \tag{2.1}$$

where $d_T(u,v)$ is the cost of the shortest path between nodes u and v in the tree. This directly corresponds to the performance of Arrow with uniformly random requests, since all pairs of nodes have the same probability of being used, meaning their distance in the tree contributes the same amount to the cost of an average request. In Section 3.2 we discuss such minimal trees further. However to calculate this value, a global view of the graph costs in needed, which nodes do not have at runtime.

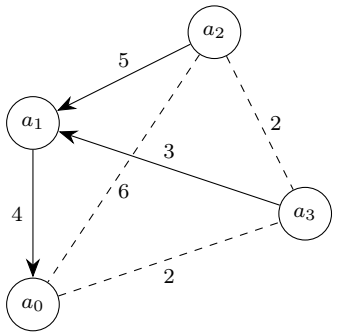
We can however devise a heuristic that minimizes pair distance as good as possible in a local subgraph consisting of only the nodes in the request path. Specifically when node a_{k+1} needs to select a new parent node out of the candidates $A_k = \{a_0, \ldots, a_k\}$, it looks at the tree graph $T = (A_k, E')$ of nodes in the request path that was constructed from the parent selections of all previous nodes in the path. It then selects the node a_i such that total pair distance is minimized, meaning $\mathcal{C}_{pairs}(T_i)$ is minimal with $T_i = (A_k \cup a_{k+1}, E' \cup (a_i, a_{k+1}))$ being the new spanning tree if node i were to be selected. See Figure 2.2 for an

example. This minimum can indeed be calculated efficiently. The main ideas of this calculation are described later in Section 3.2, which discusses an iterative tree initialization algorithm.

function SelectNewParent(A_k)
  return a_i such that $\mathcal{C}_{pairs}(T_i)$ is minimal, use highest i for tie-breaking
end function



(a) a_2 chooses a_1 as its new parent, since the total pair distance for that tree $\mathcal{C}_{pairs}(T_1) = c_{T_1}(a_0, a_1) + c_{T_1}(a_0, a_2) + c_{T_1}(a_1, a_2) = 4 + 9 + 5 = 18$ is smaller than the one for a_0 which would be $4 + 10 + 6 = 20$



(b) Even though edge (a_3, a_1) has the highest cost, a_3 chooses a_1 as its new parent, since it minimizes the total pair distance. Here $\mathcal{C}_{pairs}(T_0) = 37$, $\mathcal{C}_{pairs}(T_1) = 36$, $\mathcal{C}_{pairs}(T_2) = 38$

Figure 2.2: Local Pair Distance Minimizer example

2.6 Fixed Ratio

Hop-based Another simple heuristic is one which chooses the node at a fixed ratio $t \in [0, 1]$ between the first and the last one. As an example, with a request path a_0, a_1, a_2, a_3, a_4 and a ratio of $t = \frac{1}{2}$, node a_5 would choose a_2 as its new parent, since it's half-way between a_0 and a_4. We use the earlier node in case of a ratio being in-between two nodes. See Figure 2.3 for an example.

function SelectNewParent(A_k)
  return $a_{\lfloor t \cdot k \rfloor}$
end function

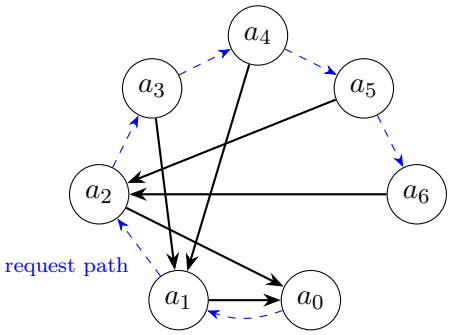


Figure 2.3: Hop-based Fixed Ratio heuristic with a ratio of $\frac{1}{2}$ and a request path of 7 nodes. Node a_6 chooses $a_{\lfloor \frac{1}{2} \cdot 5 \rfloor} = a_2$ as its new parent. A binary tree is created as a result.

Cost-based A possible variation of this heuristic doesn't use the ratio on the node counts, but on the edge costs instead. A request path $A_k = \{a_0, a_1, a_2, \ldots, a_k\}$ gets mapped to costs c_i representing how far the request had to travel to get to node a_i. These costs are defined as

$$c_i = \sum_{j=1}^{i} c(a_{j-1}, a_j) \tag{2.2}$$

The last node whose c_i is lower than or equal to $t \cdot c_k$ is then selected.

function SelectNewParent(A_k)
  return a_i with the maximum i given the constraint $c_i \leq t \cdot c_k$
end function

These heuristics have the following interesting properties:

- For $t = 0$ they are equivalent to Ivy. Similarly with $t = 1$ they are equivalent to Arrow.
- For $t = \frac{1}{m}, m \in \mathbb{N}$, the hop-based version builds an m-ary tree on the request path. See Figure 2.3 for an example of it building a binary tree with $m = 2$.
- If the costs are the same for all graph edges, the cost-based version is equivalent to the hop-based one.

2.7 Dynamic Star

To understand the Dynamic Star heuristic, we first look at the results from Peleg [8], which for one is about finding a good tree to use for Arrow. A probability distribution $p_i, i \in V$ for each node making a request is given. Then the chance that a request travels from node i to node j is simply $p_i p_j$. Now let $d_T(i,j)$ be the cost to go from node i to j in some spanning tree T, similarly let $d_G(i,j)$ be the cost for the way from i to j by taking the shortest path in graph G. Then the expected cost of a single request in a tree and the graph respectively is

$$\begin{aligned} c_R(T) &\coloneqq \sum_{i,j} p_i p_j d_T(i,j) \\ c_R(G) &\coloneqq \sum_{i,j} p_i p_j d_G(i,j) \end{aligned} \tag{2.3}$$

Now let T_c denote a shortest-path tree from center node c, meaning a tree for which every node v has a shortest path to c, i.e. $d_{T_c}(c,v) = d_G(c,v)$. The paper then shows that the T_c with lowest $c_R(T_c)$ is a 2-approximation of the costs in the graph:

$$c_R(T_c) \leq 2 \cdot c_R(G) \tag{2.4}$$

This means such a T_c is a relatively good tree to use for Arrow, since the cost of the requests isn't much greater than the best possible path they could take. And since in our case we have a complete graph and metric costs, a shortest-path tree from node c can simply be a star, connecting every other node to c directly. With our graph weights $c : E \to \mathbb{N}^+$ this then reduces the calculation of $c_R(T_c)$ to

$$c_R(T_c) = \sum_{i,j} p_i p_j (c(i,c) + c(c,j)) \tag{2.5}$$

Now this Dynamic Star heuristic measures the distribution p_i at runtime and dynamically adjusts the tree according to it, striving for a star. The fact that Arvy is a distributed algorithm however poses a problem, since nodes can't obtain a global view of the system to get accurate values for p_i, so we will have to make do with every node v having a local approximation p_i^v of p_i.

To implement this, the idea is that each node v maintains counts $n_i^v \in \mathbb{N}_0$ of how many times each node i made a request, initialized with all 0. Because v always knows how many requests it made itself, the value for n_v^v is always accurate, while all others might be outdated values. To update n_i^v, the following rules are used:

1. When node v makes a request, set $n_v^v \leftarrow n_v^v + 1$
2. Any message sent from v includes a subset of known counts $\{n_i^v, i \in S \subset V\}$, which when arriving at u will update all received values if they are bigger than the values already known: $n_i^u \leftarrow \max(n_i^u, n_i^v), i \in S$.

Depending on the choice of the subset S of value updates to send, the information propagation of the true values in the network might be slower or faster. Generally the bigger $|S|$, the faster information propagates. Here are some interesting choices for S:

- When node v makes a request for the token, it only sends along its own request count with $S = \{v\}$, while all nodes along the request path use this same S. Since the only way for the rest of the network to know the accurate request count of the requesting node is for it to send the updated value, this is one of the most reasonable single-element choices for S and is the choice that will be used for our simulations.
- With every node forwarding all its values, we get $S = V$. Propagation of values is as quick as possible, but we also have $\mathcal{O}(n)$ complexity for the size of messages.

Once such n_i^v are known, the approximated request probabilities p_i^v are then simply

$$p_i^v = \frac{n_i^v}{\sum_j n_j^v} \tag{2.6}$$

With this we have everything in place for the heuristic to select a new parent out of the available options $A_k = \{a_0, a_1, \ldots, a_k\}$. We do so by selecting the a_i with the lowest cost as the center of the star.

function SelectNewParent(A_k)
  return Choose a_i with lowest $c_R(T_{a_i})$,
   use highest i for tie-breaking
end function

Justification is needed for the fact that node a_{k+1} selecting a new parent seems to access costs not imminent to it with the calculation of values $c_R(T_{a_i})$, which was given as a restriction in the model. This can be circumvented by each node a_i calculating $c_R(T_{a_i})$ locally when the request travels through it, which is allowed since in Equation 2.5 uses only costs next to the center node a_i (this only works for star-shaped shortest path trees). Then that value can get sent in the message to all subsequent nodes in the request path, which can then use this result for the selection of the minimum. Note however that this is not exactly the same as calculating all $c_R(T_{a_i})$ values in the final node in the request path, since values for n_i^v will be different, but it's the best we can do.

Unfortunately this heuristic needs $\mathcal{O}(n^2)$ time for the calculation of $c_R(T_{a_i})$ in each node along the request path, making it very slow to run in comparison to others. Messages only need to send S and the a_i with lowest $c_R(T_{a_i})$, resulting in $\mathcal{O}(|S|)$ message size. Note that this heuristic converges to a static tree over time, which can be a problem with non-uniform request patterns. While this isn't focus of this thesis, we have included some remedies for this in Appendix B.

2.8 Graph-specific: Recursive Clique

While all previous heuristics work on arbitrary graph costs, in this section we describe a specialized one to work well on specific graphs only. As we will see in Subsection 4.3.1, Ivy has good performance on cliques of less than 5 nodes. Taking advantage of this fact, we design a graph resembling a clique, but where each node can contain another clique within, and so on. See Figure 2.4 for an example with 2 levels of cliques and each level having 3 nodes. This resulting recursive clique graph allows running Ivy on each level individually, enabling us to scale the beneficial performance from less than 5 nodes to many more.

A recursive clique graph has a level count parameter l representing the number of clique levels and a base parameter b for the number of nodes in a clique on a single level. We use the nodes in the lowest level cliques as our graph nodes and call nodes in higher levels *group nodes* since they group together graph nodes in lower levels. This gives us a total graph node count of b^l. To indicate that graph nodes in the same lower level are closer to each other, we define the costs in the layer above to be $\eta > 1$ times higher. Edges in the lowest level are defined to have cost 1, edges in the second-lowest layer then have cost η, for the next level η^2, and so on, up to edges in the top-most layer having cost η^{l-1}.

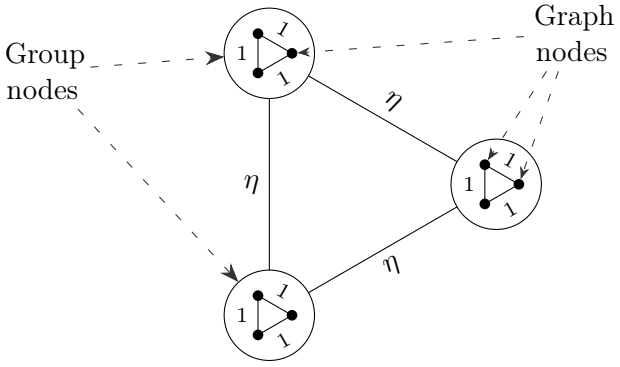


Figure 2.4: Recursive Clique graph with $l = 2$ levels and $b = 3$ nodes in each level. Graph nodes in the same group node have distance 1 between them, while graph nodes in different group nodes have distance η between them

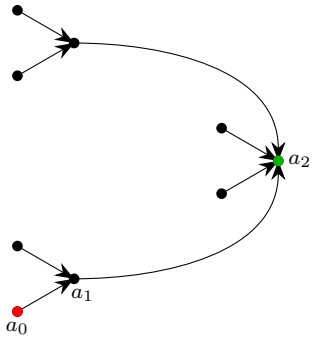
When running a recursive Ivy algorithm on such a graph, the guiding rules for selecting new parents are as follows:

1. The spanning tree needs to maintain the invariant that only at most one edge can exist between two group nodes in a single clique. This ensures that if both the requesting graph node and the one holding the token are in the same group node, the request path does not exit that group node. In addition this means that a request path can include at most $b-1$ edges of length η^{l-1} from the most upper level (and similarly for lower levels), therefore limiting the number of long edges.

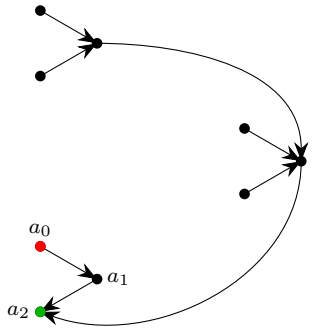
2. If multiple graph nodes on the request path satisfy the first rule, choose the earliest possible one of them as the new parent. This ensures that the inverted arrows on the request path lead future requests to the root on the fastest way possible, giving Ivy-like shortcut behavior on each clique and across levels.

See Figure 2.5 for an example of how this heuristic behaves with these rules.

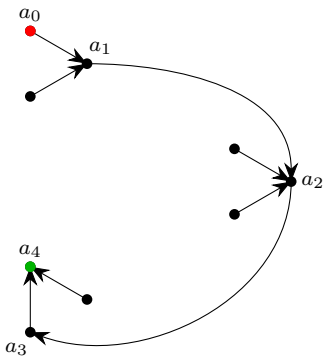
While we only limit ourselves to three parameters for simplicity, this idea can be extended much further: Each level can contain arbitrary graphs. As an example, with three group nodes at the top-most level, the first group node could contain a star graph, the second one a ring graph and the third one yet another nested graph. This then also calls for running different Arvy heuristics on each graph, using whichever works best for the respective costs, making this a very flexible approach.



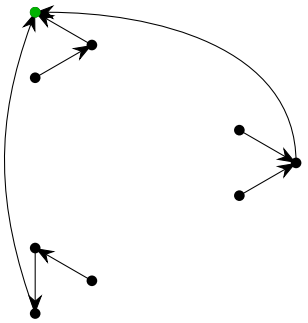
(a) The tree we start with satisfies the tree invariant, since between all pairs of three group nodes there is only at most one connection. The first graph node a_0 on the request path makes a request for the token which lies at root node a_2. a_2 will invert its arrow to point to a_0 with a long edge, which the invariant allows because the previous long edge (a_1, a_2) is discarded when the request traverses it



(b) In the new tree, a_0 makes a request for the token at a_2 in the same group node. As with normal Ivy, all request path nodes will connect back to a_0 who made the original request. Notice that since the request was made from the same group node where the token currently resides, the group node wasn't exited and no long edge was traversed



(c) With a new request, the request path has two long edges, invoking the Ivy behavior on the upper level: a_3 will create a shortcut connection back to a_0. Then when a_4 receives the request, the rules disallow it from connecting back to a_0 as well, since this would lead to multiple edges between the same two group nodes. Therefore a_4 has to connect back to a_3



(d) The resulting tree still only has 2 long edges due to how the rules were enforced

Figure 2.5: Example of the Recursive Clique heuristic on a recursive clique graph with parameters $l = 2$ and $b = 3$

Chapter 3

Graph Costs, Initial Trees and Request Sequences

The performance of an Arvy algorithm depends not only on the algorithm itself, but also on the costs of the edges, the initial tree and the request sequence. In this section we describe all variations of these to be used in the simulations.

3.1 Graph Costs

Uniformly Random d-dimensional Hypercube These costs model uniformly random points $p_i : [0,1]^d$ in a d-dimensional hypercube, using the euclidean distance between them as the cost. We will be using $d = 2$ in our evaluations.

$$c(u,v) = \|p_u - p_v\|_2 \tag{3.1}$$

Clique This is a cost function based on an underlying graph where all nodes are connected to each other with a unit cost.

$$c(u,v) = \begin{cases} 0 & \text{if } u = v \\ 1 & \text{otherwise} \end{cases} \tag{3.2}$$

Recursive Clique These costs are given by the specialized graph of the Recursive Clique heuristic, see Section 2.8.

3.2 Initial Trees

Minimum Spanning Tree The minimum spanning tree is a tree with lowest total edge cost. See Figure 3.1 for an example of such a tree.

Uniformly Random Tree A uniformly random tree out of all possible trees.

Best Star Tree As we learned in Section 2.7, the shortest-path tree with minimum $c_R(T_c)$ for some center node c has at most two times longer paths than in the graph itself. By assuming uniformly random requests with $p_i = \frac{1}{n}$ for all nodes i, we can use this to generate an initial tree and use it for unknown request distributions. This also simplifies the calculation of $c_R(T_c)$ to

$$\begin{aligned} c_R(T_c) &= \sum_{i,j} \frac{1}{n}\frac{1}{n}(c(i,c)+c(c,j)) = \frac{1}{n^2}\sum_{i,j} c(i,c) + \frac{1}{n^2}\sum_{i,j} c(c,j) \\ &= \frac{n}{n^2}\sum_{i} c(i,c) + \frac{n}{n^2}\sum_{j} c(c,j) = \frac{2}{n}\sum_{i} c(i,c) \end{aligned} \tag{3.3}$$

This means finding $c_R(T_c)$ for a single c can be done in time $\mathcal{O}(n)$, and for finding the best c with smallest $c_R(T_c)$ only $\mathcal{O}(n^2)$ time is needed. See Figure 3.1 for an example of such a tree.

Minimum Pair Distance Tree This initial tree minimizes the arguably most important measure for the Arrow algorithm: The average distance between all pairs of nodes. This is important because with uniformly random requests, every pair of nodes has the same chance of occurring subsequently in the request sequence. Consequently the average time to satisfy a request is the average distance between all pairs of nodes. See Figure 3.1 for an example of a tree that minimizes this. Noticeable is that central nodes are forming and not all short edges are used.

A way to calculate this minimum is by brute force search through all possible trees, which can be done with the Prüfer [12] sequences. Finding the total pair distance for a single tree can then be done with an algorithm similar to the one described in the next paragraph. However because there are as much as n^{n-2} possible trees [13] and finding the total pair distance takes $\mathcal{O}(n^2)$ time, this gives a total running time of $\mathcal{O}(n^n)$, which is not practical for more than a few nodes. It is very much possible for there to exist a faster algorithm for finding such a tree.

Approximated Minimum Pair Distance Tree Alternatively, we describe an algorithm that constructs a tree in $\mathcal{O}(n^3)$ time that tries to minimize total pair distance as well as possible. First let $d_T(u,v)$ denote the distance of the shortest path from u to v in a tree graph $T = (V', E')$ of nodes $V' \subset V$ and edges $E' \subset E$. Then let $\mathcal{C}_{pairs}(T)$ denote the total distance between all pairs of nodes in T.

$$\mathcal{C}_{pairs}(T) = \sum_{u,v \in V'} d_T(u,v) \tag{3.4}$$

The algorithm is based on the idea of connecting nodes one-by-one to the tree, in every step choosing the node and a connecting edge that minimizes the total pair distance. We do this efficiently by maintaining a data structure that allows querying the increase of total pair distance for a node and edge in constant time.

Let V' be the set of nodes currently included in the tree. Our data structure then encompasses an array $s : V' \to \mathbb{R}$, where $s(v)$ stores the total distance from node v to all other nodes in the current tree. In addition, we store $d_T : (V', V') \to \mathbb{R}$ in a two-dimensional array.

$$s(v) = \sum_{i \in V'} d_T(i, v) \tag{3.5}$$

Now if node $v \in V \setminus V'$ and edge (u, v) with $u \in V'$ were to get added to the tree to get $T' = T \cup (\{v\}, \{(u, v)\})$, the total pair distance increases only by the new pairs from v to all nodes in V'. With our data structure we can calculate this increase in constant time:

$$\begin{aligned} \mathcal{C}_{pairs}(T') - \mathcal{C}_{pairs}(T) &= 2 \sum_{i \in V'} d_{T'}(i, v) \\ &= 2 \sum_{i \in V'} \big(d_T(i, u) + c(u, v)\big) \\ &= 2 \left(\sum_{i \in V'} d_T(i, u) + |V'| \cdot c(u, v) \right) \\ &= 2 \big(s(u) + |V'| \cdot c(u, v)\big) \end{aligned} \tag{3.6}$$

Our algorithm then looks as follows:

1. Set $V' = \{r\}$ for an arbitrary node $r \in V$ and initialize the data structure with $s(r) = 0$ and $d_T(r, r) = 0$
2. Among all nodes $v \in V \setminus V'$ not yet in the tree and nodes $u \in V'$ already in the tree, select the pair (u, v) that increase total pair distance minimally when this edge is added, by calculating $2\big(s(u) + |V'| \cdot c(u, v)\big)$ for all of them.
3. Update s and d_T as follows:
   (a) The total distance from the new node v to all other nodes is just the total distance to all other nodes from the node u it was connected to, plus the new edge once for every node: $s(v) \leftarrow s(u) + |V'| \cdot c(u, v)$
   (b) For all other nodes, s needs to be increased by the shortest path to v, which is just the shortest path to u plus the cost of the edge (u, v): $s(i) \leftarrow s(i) + d_T(i, u) + c(u, v)$ for all $i \in V'$

(c) The shortest path from all nodes to v gets initialized to the shortest path to u plus the cost of the edge (u, v): $d_T(i, v) \leftarrow d_T(i, u) + c(u, v)$ for all $i \in V'$. The same for the the shortest path from v to all nodes.

4. Unless $V' = V$, go to step 2

Because the tree gets one more node in every iteration, the sequence for the number of edges between tree and non-tree nodes is $1 \cdot n,\ 2 \cdot (n-1),\ 3 \cdot (n-2),\ \ldots,\ (n-1) \cdot 2,\ n \cdot 1$. Every iteration therefore needs $\mathcal{O}(n^2)$ time to find the edge minimizing additional pair distance and another $\mathcal{O}(n)$ to update the data structure. The total complexity of this algorithm is therefore $\mathcal{O}(n \cdot (n^2 + n)) = \mathcal{O}(n^3)$. See Figure 3.1 for an example of a tree constructed with this algorithm.

3.3 Request Sequences

Uniformly Random Requests This request sequence distributes requests uniformly at random across all nodes. The request probability for each node v is therefore $p_v = \frac{1}{n}$, independent of previous requests.

Adversarial Requests This request sequence represents an adversary that can control which nodes issue requests, while being able to inspect the current tree. For selecting a node to request from next, it chooses the one furthest away by cost from the token in the current spanning tree.

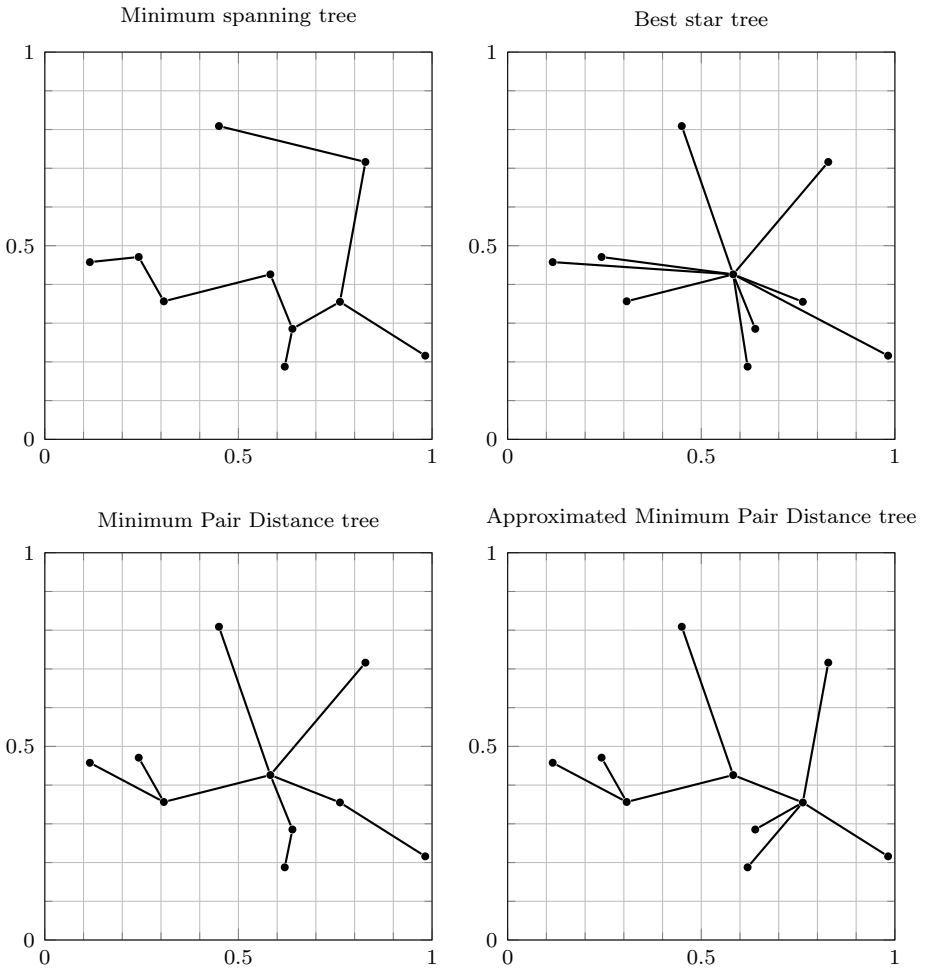


Figure 3.1: Different trees on a 10-node graph of uniformly random points in the unit square. Notable is that the Approximated Minimum Pair Distance tree is very similar to the Minimum Pair Distance tree

Chapter 4

Results

In this chapter we simulate Arvy algorithms on different *scenarios* to see how well they perform. A scenario consists of the number of nodes in the graph, what edge costs to use (see Section 3.1) and the sequence of requests (see Section 3.3). We will focus on uniformly random 2-dimensional hypercube costs since it provides interesting and realistic values, even for a low number of nodes. Similarly we will mostly use uniformly random request sequences.

We then run different Arvy algorithms from Chapter 2 with different initial trees on such scenarios and compare them against each other. Initial trees are described in Section 3.2. Since all heuristics except Arrow change the tree structure, we run them with a uniformly random initial tree as a neutral choice. Only for Arrow itself we vary the initial tree to find the best one to use.

For evaluating how well heuristics perform we look at the measures defined in Section 1.2. This includes the average request time $\mathcal{C}_{time}$ and the average request hops $\mathcal{C}_{hops}$. In addition, to get a general idea of the properties of the tree, we sometimes look at the average tree edge distance in the spanning tree $T = (V', E')$ over time, defined as

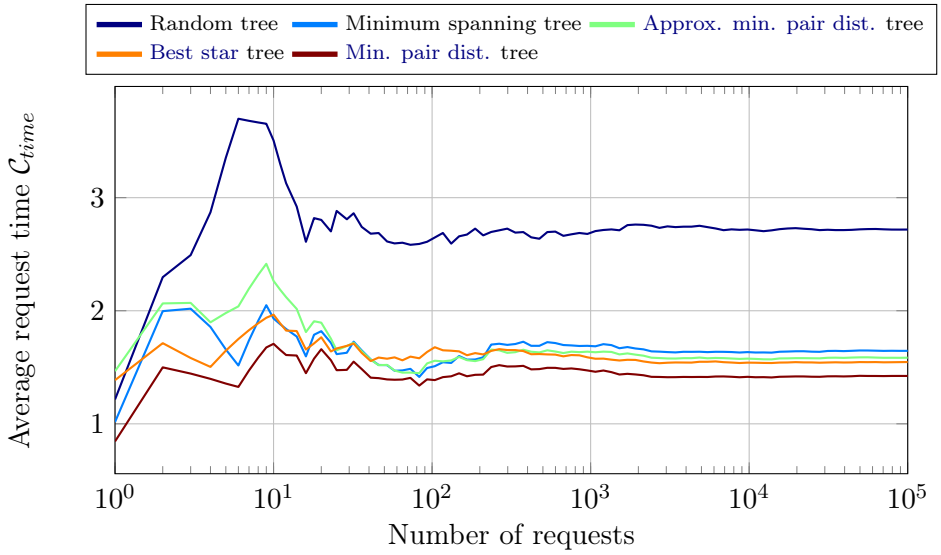
$$\mathcal{C}_{edges}(T) = \frac{1}{|E'|} \sum_{(u,v) \in E'} c(u, v) \tag{4.1}$$

4.1 Preliminary Results

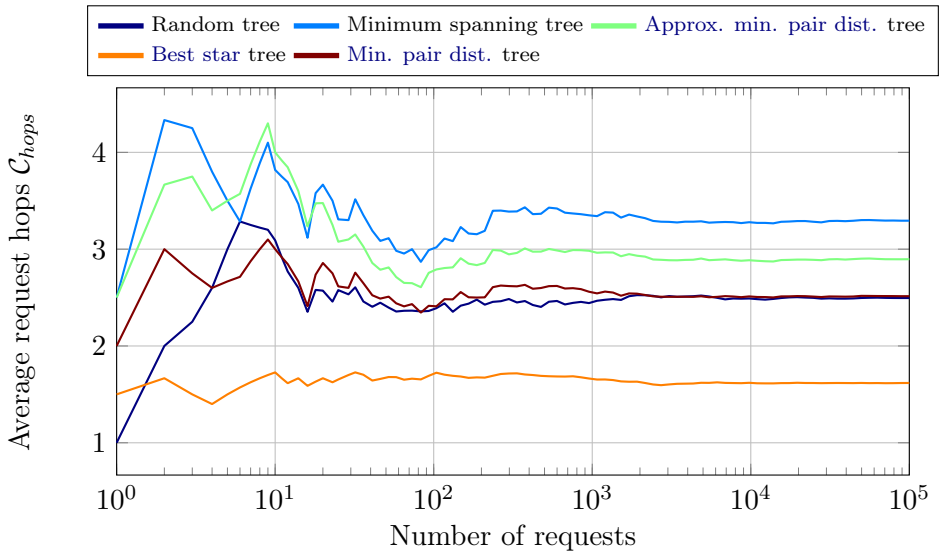
The goal of this section is to reduce the number of possible combinations of heuristics, initial trees and other parameters. In particular we want to see which tree works best for Arrow, confirm whether certain algorithms converge to a static tree and find a good value for t in the Fixed Ratio heuristics.

4.1.1 Best Initial Trees for Arrow

For this evaluation we run Arrow on different initial trees to see how well they perform. We use only 10 nodes such that we can compute the Minimum Pair Distance tree in reasonable time. For graph costs we use uniformly random points from a 2-dimensional hypercube and we will use uniformly random requests. First we look at how the average time to satisfy requests changes over time:



Not surprisingly, the tree minimizing pair distance outperforms every other tree. Unfortunately our algorithm for constructing this tree needs impractical $\mathcal{O}(n^n)$ time. Interestingly, the next best one is the best star tree with only a running time of $\mathcal{O}(n^2)$, followed closely by the Approximated Minimum Pair Distance tree with running time of $\mathcal{O}(n^3)$. From the plot we can also see that a completely random tree is understandably not a good tree to use for low average request time. Conclusively we will use the best star tree for future performance evaluations of Arrow for random requests. Now let's also look at the number of hops for each of them.

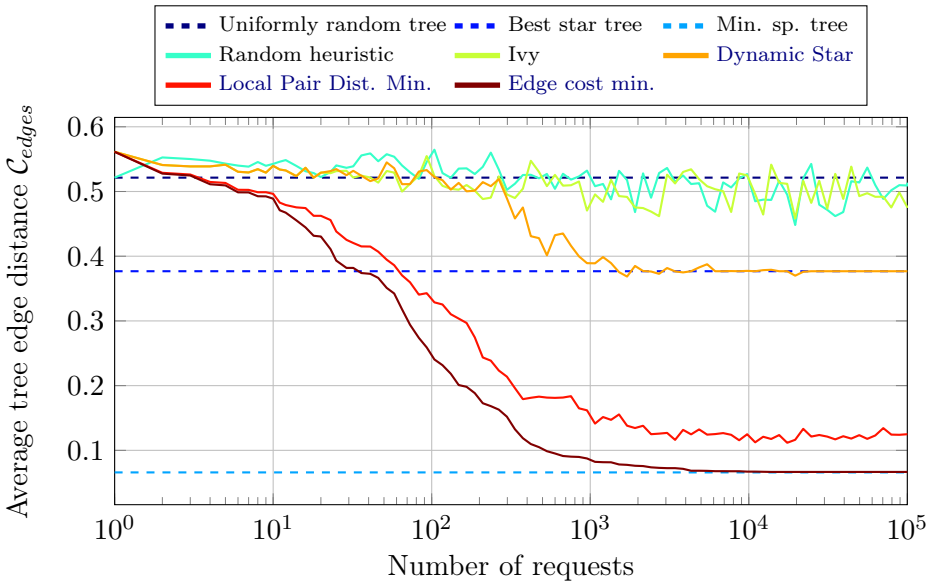


As one would expect from stars trees, they have a very low hop count. This value is lower than 2 because nodes can make requests when they have the token already, giving a hop count of 0, and the center node always has a hop count ≤ 1. Interestingly, the random tree has a decently low hop count, which is related to the results in [2, 3].

Note that while this is only the result for a single random seed with only 10 nodes, the results are similar for other configurations. Also, since in this case the tree never changes, it would be possible to calculate accurate expectations for these measures without having to rely on a simulation.

4.1.2 Tree Convergence

In this section we look at the behavior of heuristics in regards to the tree they end up with after many uniformly random requests. To get a feeling for how the tree changes over time, we look at its average edge cost. We use only 100 nodes, since the Dynamic Star heuristic is quite slow to run. For edge costs we use uniformly random points in a 2-dimensional hypercube. In addition to looking at the average tree edge cost, we also plot some fixed baselines for known trees. This includes one for the minimum spanning tree, one for the best star tree, and one for a uniformly random tree. All algorithms initially start with a completely random tree.



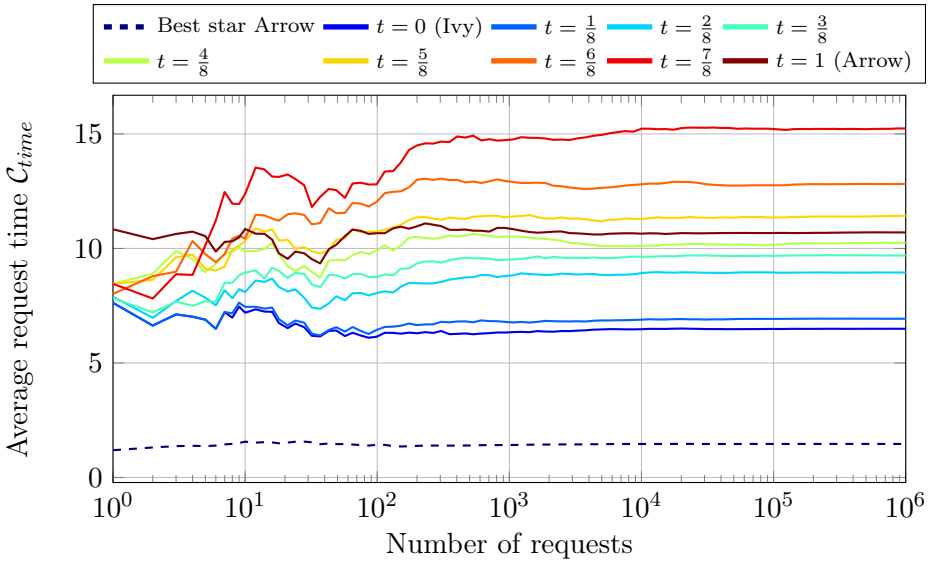
What we can see from this plot:

- The Dynamic Star heuristic converges to the best star tree, indicated by their lines merging into one. This makes sense, since they both optimize for a minimal $c_R(T_c)$ with a uniform request distribution.
- As already predicted in Section 2.4, the Edge Cost Minimizer heuristic converges to the minimum spanning tree. Here we can directly see how this algorithm only ever decreases tree edge costs, indicated by its line monotonically decreasing until it reaches the lowest possible state of the minimum spanning tree.
- The Ivy algorithm seems to continuously have a tree with high edge costs, in the same range as a completely random tree. This makes more or less sense since Ivy does not look at edge costs at all when selecting new parent nodes. The same happens for the random heuristic.
- Interestingly, the Local Pair Distance Minimizer heuristic seems to go towards a rather low average edge cost with no known baseline tree in that region.

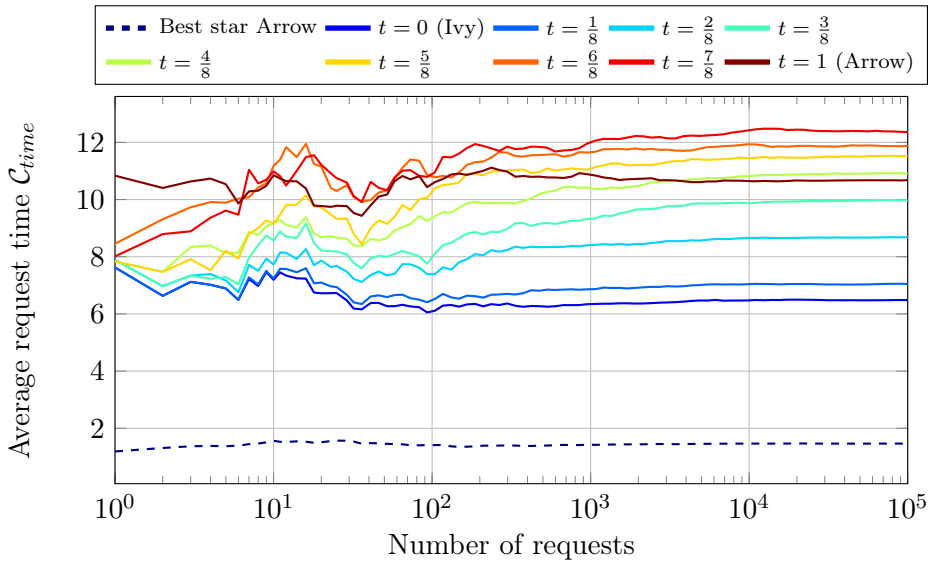
From this we conclude that for random requests, the eventual behavior of the Dynamic Star and Edge Cost Minimizer heuristics can be achieved by simply using Arrow on the best star tree and the minimum spanning tree respectively. Therefore we don't further look at these heuristics in such a scenario.

4.1.3 Fixed Ratio Heuristics

Here we look at the Fixed Ratio class of heuristics, both hop-based and weight-based ones, with different values for the ratio t, including $t = 0$ for Ivy and $t = 1$ for Arrow, with steps of $\frac{1}{8}$ between them. We will be using 1000 nodes with uniformly random 2-dimensional hypercube costs and uniformly random requests. We will be using Arrow with the best star tree as a baseline. All Fixed Ratio algorithms will start with a completely random tree.



From this we can see that this heuristic is weaker than both Ivy and Arrow with the best star for all values of t. Noticeable is that generally the higher the ratio, the worse is the performance. One exception from this is Arrow with $t = 1$, which doesn't change its initial random tree. The fact that the lines for ratios $\frac{5}{8}$, $\frac{6}{8}$ and $\frac{7}{8}$ are above the Arrow line indicates to us that such high ratios produce a tree even worse for performance than a completely random tree. We conclude that the hop-based Fixed Ratio algorithm is not worth looking into further. Instead we will try the weight-based version:



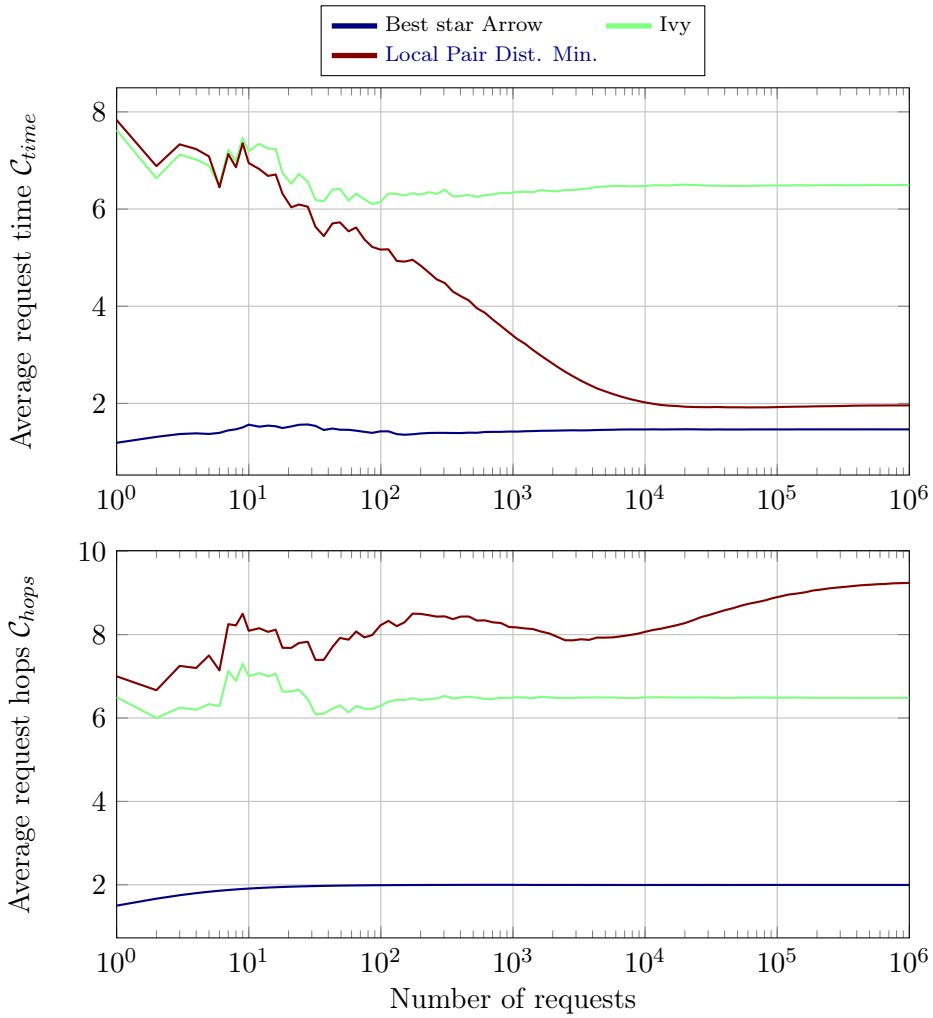
We are seeing very much a comparable pattern. We conclude that such Fixed Ratio heuristics are not fit for this task. This is perhaps surprising, since combining Arrow and Ivy in this way is very natural.

4.2 Best Heuristics

In this section we evaluate heuristics to find the best ones for different request sequences.

4.2.1 Random Requests

In this section we compare performances of different heuristics on uniformly random requests, ignoring the ones whose eventual behavior can be achieved with Arrow on a specific tree as discovered in Subsection 4.1.2 (this includes the Dynamic Star and Edge Cost Minimizer heuristics). We will again be using uniformly random points in a 2-dimensional hypercube. We use 1000 nodes.

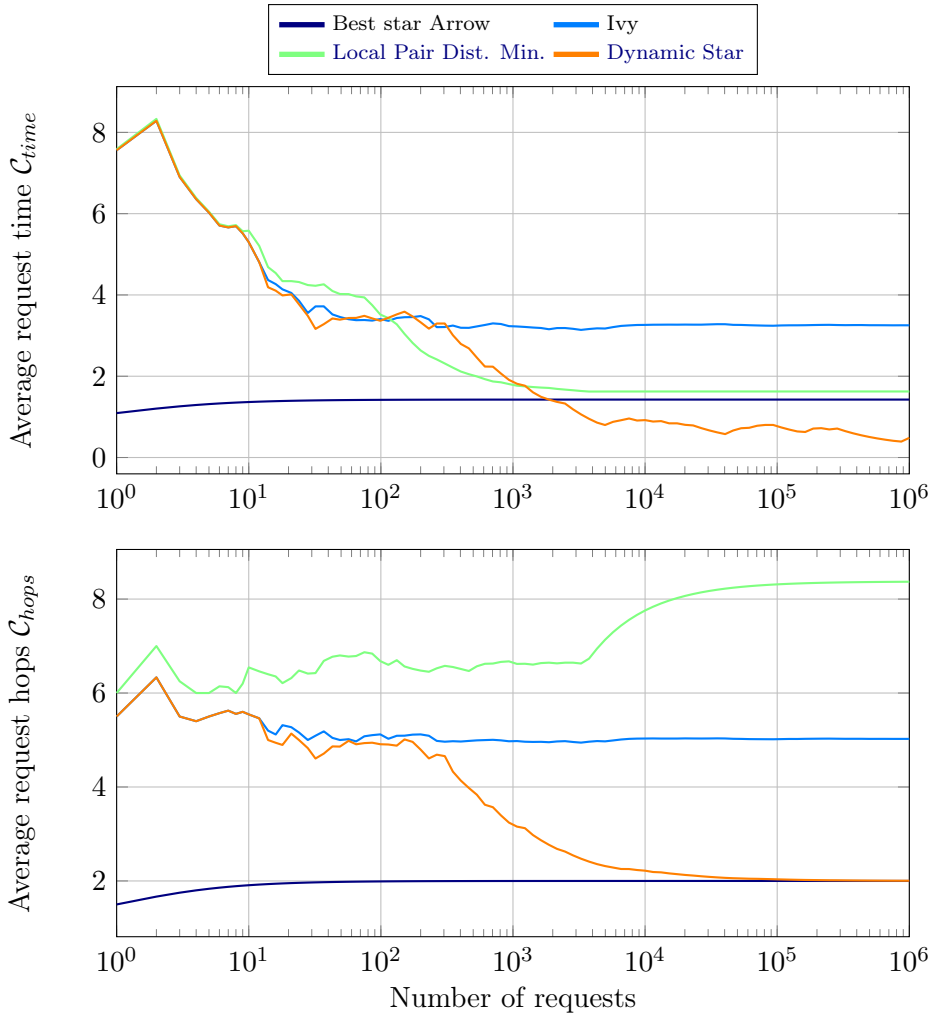


In the main plot for the average request time, we can see that Arrow clearly performs better than the other heuristics, although the Local Pair Distance Minimizer gets close to it. Ivy however is far off.

Arrow with a star tree also naturally has a lower hop count than others. Even though Ivy builds a local star on the request path, it does not come close to a global star. Noticeable is that the Local Pair Distance Minimizer heuristic has a higher average hop count than Ivy even though it performs much better time-wise. This shows that the number of hops can have very little to do with request time. Curious is the seemingly-monotonous increase in request hops from 10^4 to 10^6, associated with a very slight increase in request time, for which we don't have an explanation.

4.2.2 Adversarial Requests

In this section we look at the performance of heuristics for adversarial requests. Here we include the Dynamic Star heuristic again, since it only converges to the best star tree if requests are uniformly random and therefore doesn't show the same behavior as Arrow with that as a starting tree. Because again the Dynamic Star heuristic is rather slow to execute we only use 100 nodes. Costs are uniformly random points in a 2-dimensional hypercube.



In this adversarial request case, the Dynamic Star heuristic seems to win out as the only heuristic that adapts its behavior based on request probabilities. Seemingly it tends towards a star that allows the least possible exploitation by an adversary. In particular it appears that after about $10^{2.5}$ requests it had enough request distribution samples to find a good star center, which got accepted quickly. Arrow with the best star is decent as well, although the best star is

apparently not the best one to use in an adversarial case. The Local Pair Distance Minimizer heuristic is in third place, performing not too bad even with the highest hop count. Very peculiar is the sudden change of behavior regarding hop count at around $10^{3.5}$.

4.3 Special Graphs

In this section we focus on evaluating heuristics that work well on special graphs.

4.3.1 Ivy in Small Cliques

The fact that Arrow with the best star tree seems to have been the best heuristic on uniformly random requests so far gives the impression that perhaps Arrow can't be beaten at all. In this section we construct a counterexample by looking at Ivy in a clique with a low number of nodes. We compare Ivy with Arrow on the Minimum Pair Distance tree, since that is the best possible tree to use according to Subsection 4.1.1. After simulating 10^6 requests, the values for $\mathcal{C}_{time}$ are

Node count	Arrow	Ivy
3	1.334	1.250
4	1.499	1.443
5	1.600	1.603
6	1.665	1.739
7	1.713	1.859
8	1.749	1.963

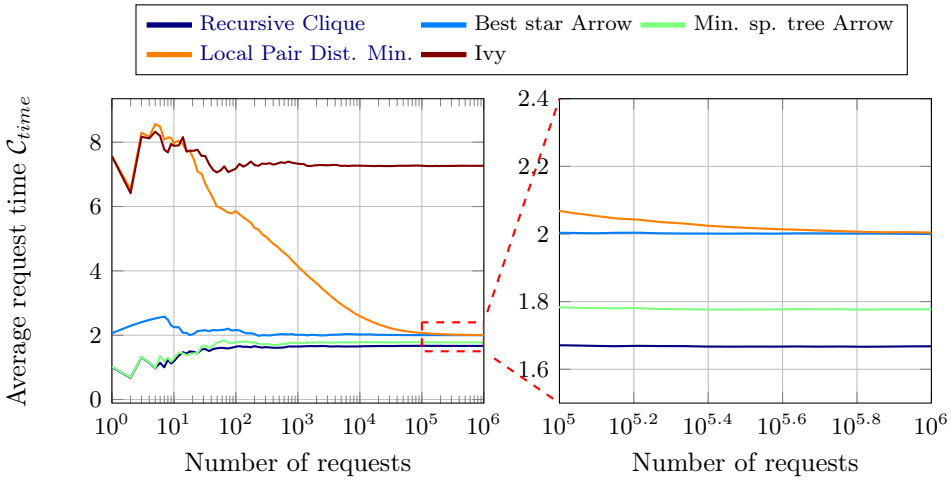
Here we can see that Ivy is in fact better than Arrow in this scenario with 3 or 4 nodes, but not with 5 (very closely) or more. This pattern is expected to continue into higher node counts. The worse performance for higher node counts is explainable with what we've discovered in Subsection 4.1.2: Ivy maintains a mostly random tree throughout, which as we've seen in Subsection 4.2.1 leads to rather poor performance in comparison to Arrow. It is perhaps surprising that this doesn't hold for node counts of 3 and 4. For 3 nodes this is however easy to see, since all requests paths with two hops in Ivy mean that the next request is guaranteed to be fulfilled in one hop due to the shortcut behavior. With Arrow however this is not the case because all two-hop request paths can be followed by another two-hop request path in the other direction, since the tree structure is maintained. This beneficial short-cutting effect of Ivy seems to outweigh the tree randomization for 4 nodes as well, which makes sense since the number of trees is so low that they can't be randomized much.

4.3.2 Recursive Clique

In this section we will measure the performance of the Recursive Clique heuristic with the hopes of it being the best heuristic to use for the type of graph it was made for. We use the parameters $l = 6$, $b = 3$ and $\eta = 5$, meaning 6 levels of 3-node cliques for a total of $l^b = 6^3 = 729$ nodes, with a cost of $\eta^{l-1} = 5^{6-1} = 3125$ between nodes in the highest level clique. We will use uniformly random requests.

We compare the performance to the usual heuristics of best star Arrow, Local Pair Distance Minimizer and Ivy. However we will add another contestant: Arrow starting with a minimum spanning tree. This is because a minimum spanning tree on such a recursive clique graph directly corresponds to a tree satisfying the invariant needed for the Recursive Clique heuristic, meaning there can be at most one edge between group nodes on the same layer, which is also beneficial to Arrow for the same reason it's beneficial to the Recursive Clique heuristic.

We look at the average request time:



Sure enough, the Recursive Clique heuristic wins over all others. The beneficial shortcut behavior of Ivy in 3-cliques is unsurprisingly maintained with multiple levels of cliques. We can also see that in this case, Arrow with a minimum spanning tree does indeed perform better than Arrow on the best star tree. Ivy is the worst heuristic in this scenario, which makes sense, since a tree constructed with Ivy can have a lot of long edges. The Local Pair Distance Minimizer heuristic seems to converge towards the performance of Arrow on the best star.

Chapter 5

Summary

In this thesis we devised some new Arvy heuristics and compared their performance. We also looked at what trees are best suited for Arrow.

With uniformly random request sequences and 2-dimensional uniformly random hypercube costs, the Minimum Pair Distance tree was found to be the best initial tree to use for Arrow. However the only algorithm we know of to construct such a tree takes $\mathcal{O}(n^n)$ time, which is impractical for many nodes. The best star tree turned out to be the next best tree for Arrow, only needing $\mathcal{O}(n^2)$ time to be constructed. The Approximated Minimum Pair Distance tree algorithm we devised turned out to be a worse tree to use in such scenarios.

Multiple new heuristics were explored:

- The Edge Cost Minimizer heuristic converges to Arrow with a minimum spanning tree.
- The Local Pair Distance Minimizer heuristic shows great potential with both uniformly random and adversarial requests.
- The Dynamic Star heuristic is the same as Arrow with a best star tree for uniformly random requests. However in adversarial requests, this heuristic turned out to be the best one to use. Unfortunately due to it needing $\mathcal{O}(n^2)$ running time in every node along the request path, this is a very slow heuristic to run in comparison to others.
- We found that Ivy is better than Arrow on very small cliques of 3 and 4 nodes. We exploited this fact for the Recursive Clique heuristic, which showed great performance on the special graphs it was designed for. The idea can be extended much further to arbitrary nested graphs as well.
- We tested Fixed Ratio heuristics based on choosing nodes a fixed ratio along the request path. Running the simulations made it clear that such an algorithm performs really poorly, with ratios near 1 even worse than Arrow on a random tree.

Bibliography

[1] P. Khanchandani and R. Wattenhofer, "The arvy distributed directory protocol," in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '19. New York, NY, USA: ACM, 2019, pp. 225–235. [Online]. Available: https://doi.acm.org/10.1145/3323165.3323181

[2] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 7, no. 1, pp. 61–77, Jan. 1989. [Online]. Available: https://doi.acm.org/10.1145/58564.59295

[3] A. Rényi and G. Szekeres, "On the height of trees," *Journal of the Australian Mathematical Society*, vol. 7, no. 4, pp. 497–507, 1967.

[4] M. J. Demmer and M. P. Herlihy, "The arrow distributed directory protocol," in *International Symposium on Distributed Computing*. Springer, 1998, pp. 119–133.

[5] F. Kuhn and R. Wattenhofer, "Dynamic analysis of the arrow distributed protocol," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004, pp. 294–301.

[6] M. Herlihy and M. P. Warres, "A tale of two directories: implementing distributed shared objects in java," *Concurrency: Practice and Experience*, vol. 12, no. 7, pp. 555–572, 2000.

[7] A. Ghodselahi and F. Kuhn, "Dynamic analysis of the arrow distributed directory protocol in general networks," *arXiv preprint arXiv:1705.07327*, 2017.

[8] D. Peleg and E. Reshef, "Low complexity variants of the arrow distributed directory," *Journal of Computer and System Sciences*, vol. 63, no. 3, pp. 474–485, 2001.

[9] K. Li, "Ivy: A shared virtual memory system for parallel computing." *ICPP (2)*, vol. 88, p. 94, 1988.

[10] D. Ginat, D. D. Sleator, and R. E. Tarjan, "A tight amortized bound for path reversal," *Information Processing Letters*, vol. 31, no. 1, pp. 3–5, 1989.

[11] R. Wattenhofer, "Lecture material in principles of distributed computing," 2019. [Online]. Available: https://disco.ethz.ch/courses/podc/exercises/solution7.pdf

[12] H. Prüfer, "Neuer beweis eines satzes über permutationen," *Archiv fur Mathematik und Physik*, vol. 27, pp. 742–744, 1918.

[13] C. W. Borchardt, "Über eine Interpolationsformel für eine Art symmetrischer Funktionen und über deren Anwendung," in *Math. Abh. der Akademie der Wissenschaften zu Berlin*, 1860, pp. 1–20.

Appendix A

Implementation Notes

As part of this thesis, a Haskell library for writing and testing Arvy heuristics was created, accessible at `https://github.com/infinisil/arvy`. In this chapter we explain some key design decisions that went into it. Most notably, Haskell's advanced type system allowed guaranteeing correctness of all Arvy heuristics defined with this library. We will look at parts of the file `lib/Arvy/Algorithm.hs` where the core of this library is defined. Needless to say, this chapter is more technical than the others and advanced Haskell knowledge is required to fully understand it.

A.1 Request Path abstraction

An Arvy heuristic is correct if nodes can only set their new parent to nodes that were previously visited on the request path. With each node being referenced as an integer, this would be a problem, because a node a_{k+1} could simply set $parent(a_{k+1}) = 0$ even though it might very well be the case that $0 \notin A_k = \{a_0, \ldots, a_k\}$, which would be an invalid choice. So instead of representing nodes as integers, we represent them as some arbitrary type with certain capabilities that reflect it being part of the request path. In particular such types will not be constructible on their own.

One such trivial capability we will need is that once we have some node index, we can forward that value via messages to other nodes. This makes sense to require, since some node a_{k+1} can't know about the possible new parents $\{a_0, \ldots, a_k\}$ unless these nodes somehow forwarded their own indices along with the messages. In Haskell, this forwarding can be represented as a typeclass on two types i_a and i_b, where i_a represents a node index earlier in the request path than node index i_b. We will state that if you have a value of i_a you can get a value of i_b, which can be done with a typeclass parametric on those types, holding a function for converting from i_a to i_b. This typeclass is trivially implementable for all same types $i = i_a = i_b$, since forwarding will then simply be the identity function.

```
-- | A class for node indices that can be forwarded between nodes
class Forwardable ia ib where
  forward :: ia -> ib

-- | All equivalent types can be trivially forwarded
instance Forwardable i i where
  forward = id
```

Now we need to somehow encode a request path in a type. In such a request path the important constraint is that messages are only sent in one direction, meaning node indices can only be forwarded in one direction as well. Looking at it from the perspective of a single node in the request path, we can forward from its predecessor to the current node and from the current node to its successor. Encoding this as a typeclass leaves us with the following for representing a node (index).

```
class ( Forwardable (Pred i) i
      , Forwardable i (Succ i)
      ) => NodeIndex i where
  -- The type representing the previous node
  type Pred i :: *
  -- The type representing the successor node
  type Succ i :: *
```

Now of course defining different types for every node along the request path is impossible, we can only use this to prove correctness. So for the implementation, we can use normal integers to represent all nodes.

```
type Node = Int

instance NodeIndex Node where
  -- Predecessors and successors in the request path are
  -- all nodes represented by integers as well
  type Pred Node = Node
  type Succ Node = Node
```

A.2 Arvy Heuristic Abstraction

The most important part of defining an Arvy algorithm is the heuristic it uses to decide the new parent node. We extend this to the notion of a *behavior*, which includes not only how new parents are selected, but also how it gets decided what a request message looks like, how the initial message is generated, and what the

final node receiving the message does. Each node on the request path should be allowed to run arbitrary code.

For representing computations we use the relatively new effect library Polysemy. In this library a computation is described with the type `Sem r a` where `r` represents the possible effects it can have and `a` being the result of the computation. As an example, a computation that's allow to use randomness and returns a boolean is described by the type `Sem '[Random] Bool`. With this said, the main type for defining an Arvy heuristic is defined as follows.

```
{- |
An Arvy heuristic for a dynamic algorithm.
- @i@ is the node index type
- @msg :: * -> *@ is the type of request messages passed between
  nodes, parametrized by the node index type for allowing messages
  to forward node indices
- @r@ is the effects the algorithm runs in, which can include
  effects parametrized by @i@ in order to allow effects dependent
  on indices -}
data ArvyBehavior i msg r = ArvyBehavior
  { -- | What message the node making the request should send to
    -- its parent
    arvyMakeRequest
    :: i -- ^ The current node index
    -> Succ i -- ^ The index of the parent node
    -> Sem r (msg i) -- ^ Returns the message to send to the parent

  , -- | Which parent an intermediate node should select when
    -- receiving a message, and what message to forward to its
    -- parent
    arvyForwardRequest
    :: msg (Pred i) -- ^ The received message
    -> i -- ^ The current node index
    -> Succ i -- ^ The parent node
    -> Sem r (Pred i, msg i) -- ^ The new parent to select and
                             -- the message to forward

  , -- | Called only for the last node in the request sequence
    -- that's holding the token
    arvyReceiveRequest
    :: msg (Pred i) -- ^ The received message
    -> i -- ^ The current node index
    -> Sem r (Pred i) -- ^ The new parent to select
  }
```

This succinct-yet-complicated type includes a lot of information. As an overview, when a node makes a request for the token, `arvyMakeRequest` is called, which generates a message to be sent to the nodes parent. Then when a message arrives at a node, depending on whether it has the token or not, the functions `arvyReceiveRequest` or `arvyForwardRequest` are called respectively, the latter of which then needs to generate another message for forwarding. Function `arvyForwardRequest` is the most interesting one since it has to handle both receiving and sending a message, so we will look more closely at its type signature.

For one it says that nodes receive messages of type `msg (Pred i)` whose values can only include node indices of predecessors to the current node in the request path. It then needs to return a value of `Pred i` as the new parent for the current node. With the way node indices are defined in the previous section, the only possible values to return are ones received from the message. This means to select a new parent, nodes have to decode the message which will contain node indices given by past nodes, then return one of them.

In addition, the function receives values `i` and `Succ i` representing the current node index and the current nodes parent respectively. In addition to the new parent node to select, the function also needs to return a `msg i` representing the message to be forwarded. The `msg` type being parametrized by `i` enforces that this message can only contain node indices to previous nodes (since all previous node indices `Pred i` can be forwarded to `i`) or the current node (which is of type `i` already). `Succ i` however can't be sent in the message because there's no way to convert it to type `i`.

Functions `arvyMakeRequest` and `arvyReceiveRequest` are special cases of `arvyForwardRequest`. The former is special in that a node making a request does not receive any message and it doesn't need to select a new parent since it becomes the root. The latter is special in that the final root node receiving the request doesn't have a parent and doesn't need to forward a message. Having separate functions for these greatly simplifies the code.

Appendix B

Non-converging Dynamic Star

In section Section 2.7 we introduced the Dynamic Star heuristic, which eventually converges to a static tree. This is because as the total number of request increases, changes to n_i^v matter less. After $R \in \mathbb{N}$ requests, another request can only cause a maximum change to p_i^v of

$$\begin{aligned}\left|p_i^v - p_i^{'v}\right| &= \left|\frac{n_i^v}{\sum_j n_j^v} - \frac{n_i^{'v}}{\sum_j n_j^{'v}}\right| = \left|\frac{n_i^v}{R} - \frac{n_i^v + \{0,1\}}{R+1}\right| \\ &= \left|\frac{n_i^v R + n_i^v - n_i^v R + \{0,1\}R}{R(R+1)}\right| = \frac{\left|n_i^v - \{0,1\}R\right|}{R(R+1)} \\ &\leq \frac{R}{R(R+1)} = \frac{1}{R+1} \to_{R\to\infty} 0\end{aligned} \tag{B.1}$$

where $\{0,1\}$ is 1 or 0 depending on whether n_i^v was incremented or not.

This is problematic because it means the longer a system runs, the less it's adapting to changes in the probability pattern. If after 10^{100} requests a request-controlling adversary shows up, it could exploit this practically static tree by constantly bouncing requests between nodes with a slow connection. If this were to happen in the beginning, it wouldn't be a problem as the heuristic could adapt to this by choosing a better star center quickly. This means the algorithm's behavior is dependent on how long it has been running. In this thesis we only look at uniform request patterns where this does not happen, but here we will shortly discuss a remedy for this.

An idea to solve this problem is to give less weight to older requests, which can be done in multiple ways:

- After every increment of n_i^v by one, all counts get decreased by a constant factor: $\forall i : n_i^v \leftarrow \alpha \cdot n_i^v$ with $0 < \alpha < 1$. If this was a global view this would work well, since after R requests an original contribution to n_i^{global} of 1 would be reduced to α^R, implementing an exponential weight falloff. However since this isn't a global view, it won't work as well, because

depending on the propagation speed with $|S|$, the n_i^v will be more outdated, resulting in old requests having more weight than they would have in a global view.

- A slight improvement over this can be achieved: For each message sent from v, the total amount of requests known $\sum_j n_j^v$ is sent along as well. When received by u, it's possible for it to calculate the difference of known requests $d \coloneqq \sum_j n_j^v - \sum_j n_j^u$. This means all n_i^u are about that many requests more in the past than n_i^v, so an update is done on them to compensate for this: $\forall i : n_i^u \leftarrow \alpha^d \cdot n_i^u$. This is better because the multiplication with α doesn't depend on the selection of $|S|$ anymore, resulting in older values being more accurately weighed.

- With the requirement of nodes having a synchronized clock it's possible to improve on this: Every fixed time step, all counts in all nodes get decreased by the factor: $\forall i : n_i^v \leftarrow \alpha \cdot n_i^v$. This is now an accurate time-based exponential falloff, independent of both the propagation speed and the request pattern.