Velocity Tuning for Air Traffic Control

Sander van der Hurk*
Supervisors: prof. dr. M.J. van Kreveld
dr. ir. A.F. van der Stappen

June 19, 2015

Abstract

In this thesis we study air traffic conflict resolution via speed control. Each aircraft is assumed to have a fixed route and no speed change during its route. We use a fast and intuitive approach to the conflict resolution which finds an exact solution using velocity tuning, a concept from robot motion planning.

The choice in priority for a pair of conflicting aircrafts introduces a binary branch in the solution space. We solve the problem using linear programming with branching constraints and reduce the search space by using a branch cutting solution method.

In this study we will reveal that the order in which the branching constraints are entered in a linear solver influences the calculation time. When using a random initialization of our branching constraints, the branch cutting solution method has outliers of a factor 100 above its average runtime. We introduce a novel method for the dynamic reordering of the decision tree, which removes these outliers. Furthermore, we show that changing the initial order of the branching constraints can have an effect similar to the dynamic reordering, and suggest and compare multiple initial sortings. Our computational study reveals that using an initial sorting based on geometrical properties of the branching constraints can result in a substantial reduction in computation time, enabling the use for real time applications.

We will run our tests using synthetic scenarios in a single-layered air sector based on CTA Amsterdam South 1. With our proposed solution method a realistically sized problem can be solved within seconds, and a extremely complicated scenario with more than 100 conflict points can be solved within minutes.

*s.e.vanderhurk@uu.nl — 3083942

1 Introduction

1.1 The problem

Air traffic control is tasked with keeping airplanes separated in the sky. During the largest portion of their flight, aircrafts have plenty of space and little work is needed to keep them separated. Because an aircraft needs to land at an airfield bottlenecks manifest around these relatively small areas, resulting in more work for the Air Traffic Controlling Officer (ATCO) to keep them separated. Two airplanes under radar control are, in general, horizontally separated with a distance of 5 nautical miles (NM) around them, and a vertical separation of 1000 feet [6]. See Figure 1. To facilitate the vertical separation, ATCOs use flight levels to indicate height on cruising altitudes instead of feet. Flight levels are spaced 100 feet, and aircrafts are only assigned to flight levels which are a multiple of 10.

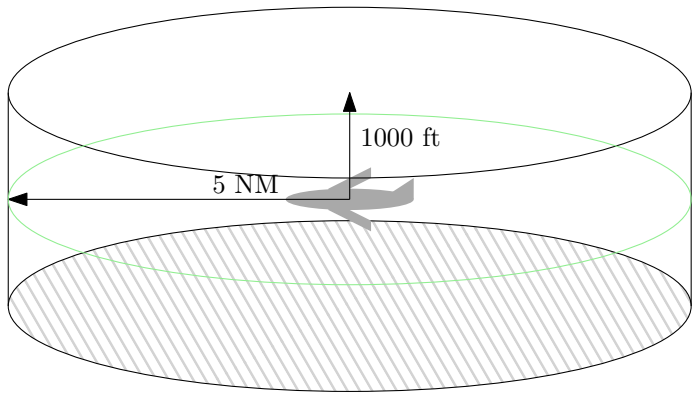

Figure 1: The volume around an airplane in which no other airplane is allowed.

The air space is getting more crowded with flights, and the number of flights is expected to grow in Europe from a current 10.0 million annual flights to 16.9 million annual flights by 2030 [12]. All these aircrafts need to be kept separated during their flight, and arrive with as little delay as possible. With the increase in traffic, the workload for the Air Traffic Controlling Officer (ATCO) will continue to grow. If two or more aircrafts need instructions in order to maintain separation, it is called a conflict resolution. There is a maximum number of conflict resolutions that an ATCO can handle within a certain amount of time. This means that an air space is limited in its capacity, amongst other factors, by the quality of the conflict resolution of the ATCO. Each conflict resolution will require mental capacity of the ATCO, and mental

capacity is limited. According to the Eurocontrol 2013 report [11], 92.7% of the air traffic delay in The Netherlands was caused by a shortage of capacity or staffing. In other countries these numbers are equally high. European delays are estimated to have cost 2.2 billion euro in 2013 alone. To help with air traffic control, and increase capacity, computer systems are needed to help with conflict detection and resolution, which will reduce the amount of mental capacity needed per conflict resolution, or even improve the quality of the chosen resolutions.

1.2 Related work

Several studies have been conducted towards aiding systems and air traffic control automation. Many of the early papers focused on trajectories but in the last two decades a shift to speed adjustment can be noticed [2,4,8]. Early work included research into neuro control and fuzzy control [7], in an attempt to transfer the expert knowledge of the air traffic controllers into a neural network. Later work focused more on the conflict resolution by calculating the problem areas. In order to save calculation time, these algorithms tend to use heuristic methods such as potential field [3] or genetic algorithms [1] to adjust the speed and route. The PHARE programme [9] solves the speed via exact calculations, but only for one aircraft at the time, considering all other aircrafts to maintain their current speed. A decision support system that optimized speed for all aircraft simultaneously using MILP was explored by Vela et al. [14]. Their system could handle peaks of 30 aircrafts within 10 minutes calculation time on a modern quadcore computer and used a time limit of 10 minutes. Not every scenario could be fully calculated in these 10 minutes, so after these 10 minutes had passed the program reported which aircrafts it could not give a conflict free speed to, and it would be up to an ATCO to make the final solution. It did show that the capacity could be greatly increased while using just a single flight level.

Air traffic will always have conflicts. To resolve these conflicts an ATCO can change altitude, direction and speed of an aircraft. Currently whenever a delay is needed for an aircraft, a common choice by air traffic controllers is the use of fanning, the practice of offsetting the heading of the aircraft to the left by a few degrees and after a short while letting it turn to the right, or vice versa. This fanning will increase the distance the aircraft will have to fly to the airport, effectively delaying it. The advantage of this is that the air traffic controller can easily see that his traffic stays separated. Two mayor disadvantages are the increase in monitoring time needed from the air traffic controller and a negative effect on the environment [14]. Studies have found that it is hard for ATCOs to see the consequences of subtle speed

changes [1]. This makes it easy to understand why an ATCO would prefer to resolve a conflict using altitude or direction adjustment instead of speed adjustment. However, these speed adjustments have proven to be highly effective for solving conflicts [13, 14].

The PHARE programme makes the information the algorithm gives very visible [9]. It has infeasible regions displayed as blobs, and shows what speed changes will result in, as can be seen in Figure 2. This system mainly allows the ATCO to see what a change will result in, but will only allow one aircraft to be changed at the time. It maintains the mental image of the ATCO but it does not give a complete solution. Vela et al. [14] on the other hand use a system of formulas, based on time separation between aircraft. This yields solutions in which the ATCO will not be able to see why the choices have been made, and thus failing to help the ATCO with his or her mental image.

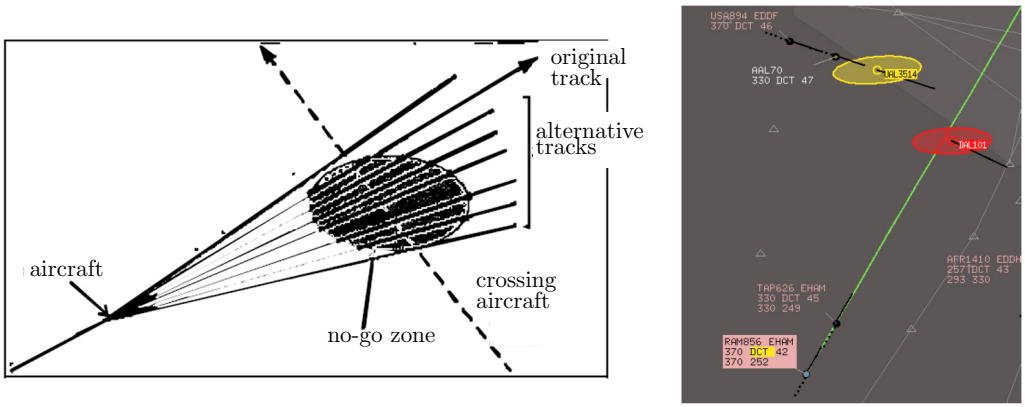

Figure 2: Left: concept of alternative tracks and the no-go zone with crossing traffic. Right: No-go zone blobs as displayed in the PHARE programme. Images from [9].

1.3 Our focus

We will look into a speed-based decision support system with an intuitive approach in order to easily visualize the problem and solution. This will allow the ATCO to remain involved in the decision making.

The first choice we make is that our model will focus on speed. An ATCO has three different ways of controlling the aircrafts, namely by changing altitude, direction and speed. The reason we choose speed is because it has proven to be very effective for solving conflicts, and because speed is the toughest of the three for an ATCO to mentally visualize.

The second choice we make is to focus on the controlled air space called the control area. The air space is divided into multiple sections, see Figure 3. In order from nearest to furthest away from an airport we find the control zone (CTR), terminal control area (TMA), control area (CTA), and upper control area (UTA). The CTR has no radar separation and can therefore not benefit from air traffic control purely based on speed. The TMA has too little space to effectively separate aircrafts purely based on speed [8]. We will focus on the CTA, which can highly benefit from air traffic control based on speed. It has a high concentration of traffic but with a larger area than the TMA, and small changes in speed have large effects in separation [1]. The CTA also provides multiple types of traffic such as crossing, descending, and ascending traffic, which will not be found in UTA. We will base one of our tests on the Dutch CTA South. According to an interview with the LVNL, there are on average 5 aircrafts at the same time in CTA South, with peaks reaching 15 aircrafts[1]. Based on the amount of time an ATCO has before a coordinated aircraft enters his air space, we choose 5 minutes as a realistic goal for solving a minimum of 15 aircrafts. Because a consumer's desktop is not a good indication, we are also interested in the trend the data will show, and therefore we will not set a time limit on our program.

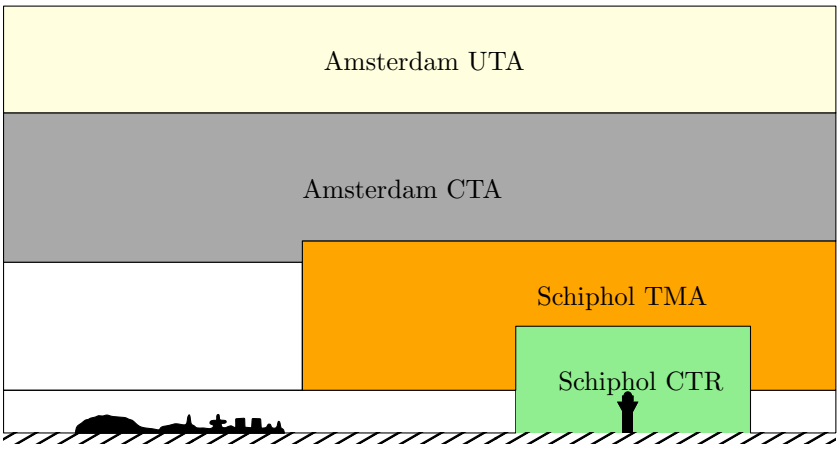

Figure 3: A simplified cross section of the different areas of controlled airspace around an airport

The third choice we make is that we will focus on finding an exact solution instead of using heuristics. The main reason listed for choosing a heuristic approach is reducing computation time [2]. Since the solution approach by

[1]Interview on 14 April 2014 with LVNL Senior Human Factor Consultant.

Vela et al. [14] is already considered a real time solution, there is no reason to reduce computation time by using heuristics. At the moment, not many aids are implemented at the Dutch air traffic control (LVNL). When enquired about the lack of decision support systems, the main reason given was that most systems do not show why certain decisions are made. This excludes the ATCO from the solution, and makes it impossible to form a mental image. This mental image is necessary for the ATCO in case of needed adjustments or system failure. We will base our model on geometric information in order to more easily clarify the decisions made by the model, helping the ATCO form a mental image of the chosen solution. To solve our problem, we will use velocity tuning as described by Lavalle [10]. This basically gives us the problem of choosing which aircraft goes first past each intersection. Guy et al. [5] solved a speed-based collision avoidance problem using geometric information and velocity tuning as well, but choose from the start which agent goes first past an intersection. We will not reduce our solution space by making such choices.

Finally, we choose to study two methods to reduce the search space of our problem. Solving our problem with velocity tuning creates a branching linear program. Vela et al. [14] have a similar branching, but do not mention how they enter their constraints in a linear solver. We will study a branch cutting solution method, which limits the search space. We expect the initial sorting of the branching constraints to have influence on the runtime of the branch cutting solution method, and will therefore initialize the solution method with different sortings of the branches. To further test the influence of the ordering of the branches with branch cutting, we will introduce a dynamic re-sorting of the branches which we will call the swap method.

1.4 Thesis organization

We will first, in Section 2, describe the problem definition and our goal. Thereafter we will define the simplified airplanes, named bodies, and describe the other simplifications and assumptions of the model.

The concepts velocity tuning and solution space will be explained in Section 3, where we will also introduce the concept of the infeasible region. We continue by showing how the infeasible region has the shape of a rotated ellipse, and explain how to compute its parameters. After mentioning the non-standard cases for conflicting bodies, wedges are introduced which will result in binary choices in our search space. Lastly, it will also be illustrated how to visualize the area where two bodies can be in conflict.

Next, in Section 4 the conversion of the model to linear constraints will be given, along with a support for choosing the sum of all velocities as our

goal function. Thereafter, we will mention a possible visualization for the solution as given by the linear solver.

Section 5 explains how to explore the entire search space, and contains two different approaches to reducing the search space. We first discuss a method for early detection of infeasible solution choices, and then we introduce our dynamic reordering of the branching linear constraints. Finally, we will mention the four different initial sortings of the branching linear constraints.

In Section 6 we will describe the test scenarios for testing the different optimizations. We will continue in Section 7 with listing our test methods, and showing the consistency of our time measurements. Then, we will examine which factors and variables influence the runtime when finding a solution. Thereafter, we will compare the runtime of the different solution methods. We end the section by exploring the performance of the fastest solution method, discuss the runtime for unsolvable scenarios, and evaluate how one of the sorting methods could be designed better.

Lastly, Section 8 will have our conclusions and suggestions for future work.

2 Forming a model

In this section we will start by giving the problem definition and goal. Thereafter, we define our simplified substitution for an airplane, named body. We will introduce the notation for the different variables, as well as shortly motivate certain simplifications.

2.1 The goal of the model

The problem we will be solving is as follows: Let there be n aircrafts in a two-dimensional plane with a starting position and a goal. For each aircraft select a speed that guarantees the aircraft stays separated from all other aircrafts until it reaches its goal. To keep the ATCO in the loop of the decisions made by the speed assistance model, we want to be able to visualize the reasoning behind the decisions, similar to the PHARE [9] system. To do this we want to base our decisions on geometric information, as was previously done by Guy et al. [5].

Our performance goal comes from the interview with the LVNL as mentioned in Section 1.3. We want to find a solution for 15 airplanes, in an air space with a size equal to CTA South, within 5 minutes on a modern hexa-core computer. Because this computer is not representative for the

computing power an air traffic control center would be able to use, we will not set a time limit on our solution method, but also look at the trend the data shows after we passed the 5 minute goal.

2.2 Bodies

We start by simplifying the airplanes in our model. To avoid assumptions about the properties of our airplanes, we will not use the word airplane from here on forward, but refer to our simplified substitutes for an airplane as a body instead.

We will now continue to define a body. A *body* is a point in a 2 dimensional plane. Bodies are not allowed to come within a certain distance of each other. Body i, denoted by B_i, will have a *separation distance* D_i to other bodies, with D_i effectively creating an empty circle around B_i with a radius of D_i, as illustrated in Figure 4. If B_i and B_j have a Euclidean distance of at least D_i, we call the bodies *separated*. For convenience, we will assume in the rest of the thesis that, for each pair of bodies B_i and B_j, there is a single minimum separation distance D where $D = D_i = D_j$. If unequal separation distances for bodies are needed, D has to be replaced with $\max(D_i, D_j)$.

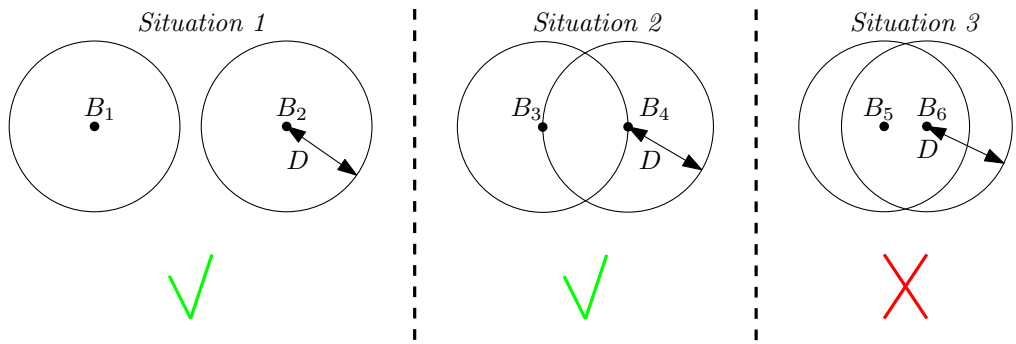

Figure 4: Separation of bodies. Only B_5 and B_6 are not separated.

Each body B_i has a *path* P^i, a directional line-segment from their *starting point* P^i_S to their *endpoint* P^i_E with $P^i_S, P^i_E \in \mathcal{R}^2$. For each pair of bodies B_i and B_j, the distance between P^i_S and P^j_S is at least D. If not, the bodies would not start separated, making it impossible to keep them separated. The distance between P^i_E and P^j_E does not need to be at least D. This is because bodies will be no longer considered for separation once reaching their endpoint, which makes it possible that one body may reach the endpoint before the other body does. We focus on straight routes. Vela et al. [14]

handle piecewise straight routes, which are not allowed to bend in conflict areas. The handling of piecewise straight routes can be easily added to the model.

Each body i has a *minimum speed* V^i_{Min} and *maximum speed* V^i_{Max}. A speed will be assigned to a body at the start of its path, which it will adopt instantaneously.

3 Using velocity tuning

In this section we will start by describing velocity tuning and introducing state space and infeasible region. Then, we will show that, if there is a non-zero angle between their paths, the shape of every infeasible region between two bodies is a 45 degrees rotated ellipse, and explain how to compute the parameters for the ellipse. Thereafter we will cover how to compute the infeasible region for special cases, such as parallel paths. We continue by introducing the concept of wedges, and finish this section by illustrating how to visualize the infeasible region in the two-dimensional plane.

3.1 Velocity tuning basics

Velocity tuning is described by Lavalle [10] as a decoupling of path planning and motion timing. Since the bodies in the model described in Section 2 have predefined paths, the only focus needed is the motion timing. We will explain velocity tuning based on two bodies.

Because velocity tuning does not include the path planning, the position of body B_i can be reduced to a one-dimensional parameter. The position of the body on the path can be described by τ, the distance B_i traveled along its path P^i, where $\tau \in I = [0, \|P^i\|]$ and $\|P^i\|$ is the length of P^i. Contrary to the description in Lavalle, we do not scale this domain back to [0,1]. Leaving this range unscaled gives us geometric properties of the infeasible region which will be used later on in Section 3.2. The position of B_i in the two-dimensional plane when given τ can be described by the function $B_i(\tau)$

$$B_i(\tau) = P^i_S + \tau H_i$$

where H_i is the unit vector describing the direction of the path P^i. The position for B_j will be referred to as $\sigma \in J = [0, \|P^j\|]$. A *state* $s = (\tau, \sigma)$ represents the situation where B_i is at position $B_i(\tau)$ and B_j is at position $B_j(\sigma)$.

The *state space* $\mathcal{S}^{ij}$ is a 2 dimensional space $\mathcal{S}^{ij} = I \times J$ containing all possible states s for B_i and B_j. The origin of the state space represents B_i

and B_j at their respective starting points. Because we defined in Section 2.2 that the distance between P_S^i and P_S^j is at least D, B_i and B_j are separated at the $s = (0, 0)$. The *infeasible region* $\mathcal{S}_{\text{Obs}}^{ij}$ is a subset of $\mathcal{S}^{ij}$ containing all $s \in \mathcal{S}^{ij}$ where B_i and B_j are not separated.

In general, a solution to velocity tuning can be found by constructing a path, originating from the origin, through the state space which keeps clear of the infeasible region. When taking into account that the bodies in the model are unable to move backwards or stand still, this path needs to be strictly increasing with respect to both axes. Because bodies are not considered for separation whenever they reach their endpoint, the strictly increasing path needs to reach either a state where $\tau = \|P^i\|$ or $\sigma = \|P^j\|$.

A path through the state space will gives us the relative speeds of B_i and B_j. The bodies will be assigned a single speed at the beginning of their path, meaning that we need to find a single ratio between the speed of B_i and B_j. To find this ratio, a line l needs to be found with *solution angle* θ where $0 < \theta < 90$ and l does not intersect with $\mathcal{S}_{\text{Obs}}^{ij}$. We choose θ to be the angle between l and the horizontal axis. In Figure 5, the possible ranges within which θ can be chosen are marked green and the range in which l would intersect with $\mathcal{S}^{ij}$ is marked red.

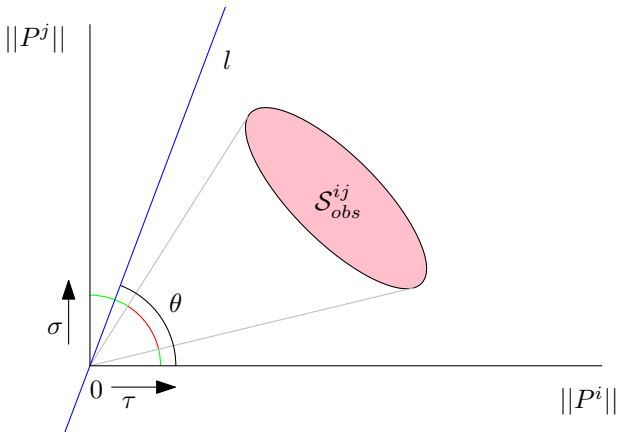

Figure 5: State space for B_i and B_j with infeasible region $\mathcal{S}_{\text{Obs}}^{ij}$ and line l with angle θ.

3.2 The infeasible region: A 45 degrees rotated ellipse

To be able to determine the possible ranges for solution angle θ the geometric properties of the infeasible region $\mathcal{S}_{\text{Obs}}^{ij}$ need to be computed. For the

computation of $\mathcal{S}^{ij}_{\text{Obs}}$, we will assume that there is a non-zero angle between the paths P^i and P^j and that $\mathcal{S}^{ij}_{\text{Obs}}$ is not an empty set. We will also assume that there is no state in $\mathcal{S}^{ij}_{\text{Obs}}$ containing either P^i_E or P^j_E. We will explain in Section 3.3 how to compute the infeasible region for cases where there P^i and P^j are parallel, and for cases where a body at his endpoint can be in conflict with another body.

In this section, we will show that $\mathcal{S}^{ij}_{\text{Obs}}$ has the geometrical properties of an ellipse in the state space $\mathcal{S}^{ij}$ for every pair of paths P^i and P^j where P^i and P^j have a non-zero angle α between them. Thereafter, we will show that the angle φ, the rotation of the ellipse, equals 45 degrees regardless of D and α. Then, we will explain how to determine a and b, the length of the semi-major axis and semi-minor axis of the ellipse. The semi-major axis is half the major axis, the longest axis of the ellipse, and the semi-minor axis is half of the minor axis. See also Figure 6. The intermediate steps for finding equations 1, 3, and 4 can be found in Appendix A.

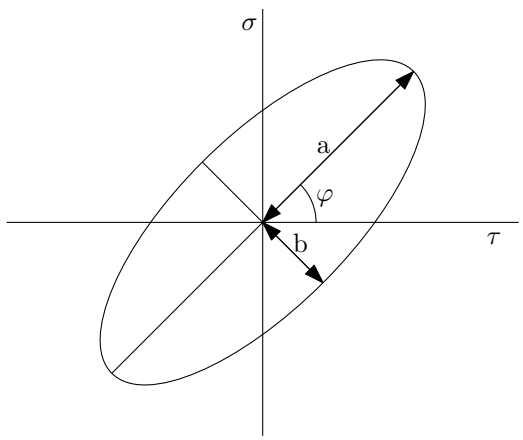

Figure 6: Generic ellipse

Given two intersecting paths P^i and P^j with angle α between them, where α is in the range $\langle 0, 90]$. By viewing P^i and P^j as lines, instead of line segments, where σ and τ are in the range $\langle -\infty, \infty\rangle$ P^i and P^j will have an intersection. The pair can be rotated and translated so that P^j aligns with the horizontal axis and the intersection between P^i and P^j is at point $(0, 0)$ without loss of generality. The implicit function for distance between a point on P_i and P_j will be used the compute the points with a distance of D. P^i can be parametrized as $(\tau\cos(\alpha), \tau\sin(\alpha))$ and P^j can be parametrized as $(\sigma, 0)$, see Figure 7. The points σ and τ with a distance of D are $(\sigma - \tau\cos(\alpha))^2 + (0 - \tau\sin(\alpha))^2 = D^2$, which can be rearranged to

equation 1.

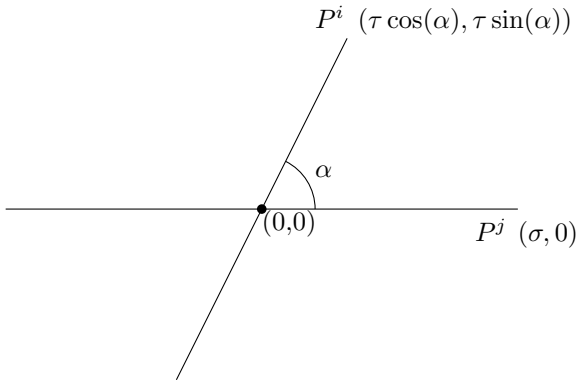

Figure 7: P^i and P^j with angle α

$$\sigma^2 - 2\cos(\alpha)\sigma\tau + \tau^2 - D^2 \quad = \quad 0 \tag{1}$$

We will now show that equation 1 is the equation for an ellipse or circle, independent of α. The general form of ellipse is given as

$$A\tau^2 + B\tau\sigma + C\sigma^2 + D\tau + E\sigma + F = 0 \tag{2}$$

with

$$\begin{aligned}
A &= a^2(\sin\varphi)^2 + b^2(\cos\varphi)^2 \\
B &= 2(b^2 - a^2)\sin\varphi\cos\varphi \\
C &= a^2(\cos\varphi)^2 + b^2(\sin\varphi)^2 \\
D &= -2A\tau_c - B\sigma_c \\
E &= -B\tau_c - 2C\sigma_c \\
F &= A\tau_c^2 + B\tau_c\sigma_c + C\sigma_c^2 - a^2b^2
\end{aligned}$$

where (τ_c, σ_c) is the center of the ellipse, which we chose to be $(0, 0)$, a and b are the length of the semi-major axis and semi-minor axis, and φ is the rotation of the ellipse, see Figure 6. For the classification of the conic described in equation 2 to be an ellipse or circle, the discriminant $B^2 - 4AC$ needs to be lesser than 0. In equation 1 $A = 1$, $B = -2\cos(\alpha)$, and $C = 1$. So the discriminant is

$$B^2 - 4AC = (-2\cos(\alpha))^2 - 4 * 1 * 1 = \cos(\alpha)^2 - 1$$

Because α is in the range $\langle 0, 90]$, $\cos(\alpha)^2$ is in the range $[0, 1\rangle$. This results in the determinant being in the range $[-1, 0\rangle$, and therefore always less than 0, and thus an ellipse or circle.

Having shown that equation 1 is the equation for an ellipse, or circle, we will now show that the ellipse is rotated by 45 degrees. When matching equation 1 to the general form of the equation for an ellipse, it can be seen that A and C are equal. If $A = C$ then

$$a^2(\sin\varphi)^2 + b^2(\cos\varphi)^2 = a^2(\cos\varphi)^2 + b^2(\sin\varphi)^2$$

and thus

$$a^2 = b^2 \vee \varphi = 45$$

The intermediate steps can be found in Appendix A. If $a^2 = b^2$, then $B = 2(b^2 - a^2)\sin\varphi\cos\varphi = 2(0)\sin\varphi\cos\varphi = 0$. In equation 1 we see $B = k * -2\cos(\alpha)$, with k as a non-zero constant, which is only 0 when $\alpha = 90$. Therefore, when α is in the range $\langle 0, 90\rangle$, $\varphi = 45$. If $\alpha = 90$ then $a = b$ which makes the ellipse a circle for which φ is irrelevant.

Now that we know that equation 1 is the equation for an ellipse with a rotation of 45 degrees, we can determine a and b. With $\varphi = 45$ and $\tau_c = \sigma_c = 0$, we get the following equations for A, B and F.

$$\begin{aligned} A &= \frac{1}{2}a^2 + \frac{1}{2}b^2 \\ B &= b^2 - a^2 \\ F &= -a^2b^2 \end{aligned}$$

Because, in equation 1, σ^2 has a factor of 1 we use the following form of the formula for an ellipse

$$\tau^2 + (B/A)\tau\sigma + \sigma^2 + F/A = 0$$

making the equations equal. We can now say $\frac{B}{A} = -2\cos(\alpha)$, and $\frac{F}{A} = -D^2$. Solving for a^2 and b^2 gives us:

$$a^2 = \frac{\frac{1}{2}D^2}{1 - \cos(\alpha)} \tag{3}$$

$$b^2 = \frac{\frac{1}{2}D^2}{1 + \cos(\alpha)} \tag{4}$$

With equations 3 and 4 we can construct the implicit function from equation 2 for the infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$ when α and D are given.

3.3 Special cases

We will explain in this section how to compute the infeasible region for cases where there P^i and P^j are parallel, and for cases where a body at its endpoint can be in conflict with another body. Whenever a body at its endpoint can be in conflict with another body the infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$ is the intersection of $\mathcal{S}^{ij}$ and $\mathcal{S}^{ij}_{\text{Obs-full}}$, where $\mathcal{S}^{ij}_{\text{Obs-full}}$ is the infeasible region if σ and τ are in the range $\langle -\infty, \infty \rangle$. When P^i and P^j are parallel, and B_i and B_j have a possibility of not being separated, $\mathcal{S}^{ij}_{\text{Obs}}$ is a slab. We will show with an example where B_i and B_j travel in opposite direction, see also Figure 8. Changing the direction of a path will result in a mirroring of the state space. Let state $s_1 = (\tau_S, \|P^j\|)$ be the first point on P^i where the distance between B_i and P^j_E equals D. Because P^i and P^j are parallel, at any state $s = (\tau_S + \epsilon, \|P^j\| - \epsilon)$ the distance between B_i and B_j equals D. This means that one boundary of $\mathcal{S}^{ij}_{\text{Obs}}$ is parallel to the line $\sigma = -\tau$. Let state $s_2 = (\tau_K, \|P^j\|)$ be the second point on P^i where the distance between B_i and P^j_E equals D. For any state $s = (\tau_K + \epsilon, \|P^j\| - \epsilon)$ the distance between B_i and B_j equals D. This means that the other boundary of $\mathcal{S}^{ij}_{\text{Obs}}$ is parallel to the line $\sigma = -\tau$. Being bound by two parallel lines, the infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$ is a slab.

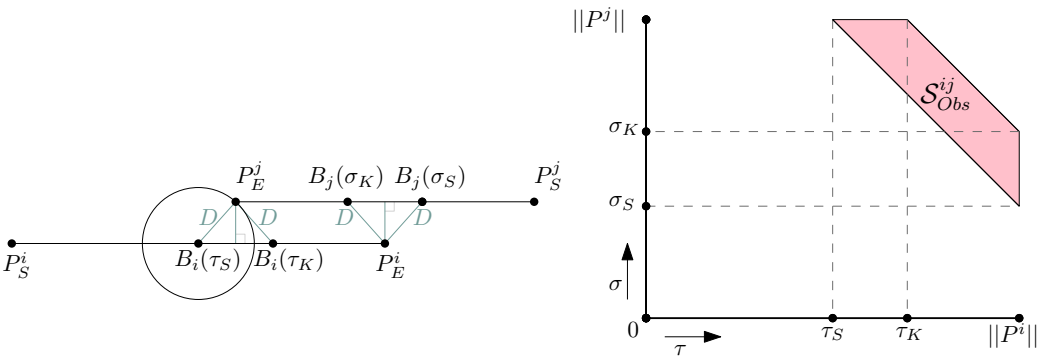

Figure 8: Left: P^i and P^j are parallel. Right: The corresponding state space $\mathcal{S}^{ij}$ with the infeasible as a slab.

3.4 Wedges

The possible range within which angle θ can be chosen to keep bodies B_i and B_j separated is affected by the infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$. We define the *tangent angle* as the angle for θ for which line l is a tangent to the infeasible region contained within the state space. By limiting the tangent angle to the infeasible region contained within the state space, the range for the tangent

angle is $[0, 90]$. Tangent angle R^{ij}_{GT} is the largest angle where l is tangential to the infeasible region within the state space. Tangent angle R^{ij}_{LT} is the smallest angle where l is tangential to the infeasible region within the state space. If there is no infeasible region within the state space, $R^{ij}_{GT} = 90$ and $R^{ij}_{LT} = 0$.

The possible range for θ is not just defined by $\mathcal{S}^{ij}_{\text{Obs}}$. There are certain states s in state space $\mathcal{S}^{ij}$ which can only be reached if either B_i moves below its minimum speed, or B_j moves above its maximum speed. Due to this minimum and maximum speed θ will always be bound by two *implicit angles*, R^{ij}_{IMin} and R^{ij}_{IMax}, regardless if $\mathcal{S}^{ij}_{\text{Obs}}$ exists or not. R^{ij}_{IMin} and R^{ij}_{IMax} are defined as follows.

$$\begin{aligned} R^{ij}_{IMin} =& V^j_{Max}/V^i_{Min} \\ R^{ij}_{IMax} =& V^j_{Min}/V^i_{Max} \end{aligned}$$

R^{ij}_{IMin} is the angle indicating the states reached when B_i moves at its minimum speed, and B_j moves at its maximum speed. R^{ij}_{IMax} is the angle indicating the states reached when B_i moves at its maximum speed, and B_j moves at its minimum speed.

The infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$ can divide the possible range for θ in up to two ranges, see in Figure 5. We will call these ranges *wedges*. Wedges W^{ij}_{GT} and W^{ij}_{LT} are defined as follows.

$$\begin{aligned} W^{ij}_{GT} =& [R^{ij}_{GT}, R^{ij}_{IMin}] \\ W^{ij}_{LT} =& [R^{ij}_{IMax}, < R^{ij}_{LT}] \end{aligned}$$

Intuitively, W^{ij}_{GT} is the wedge that is above $\mathcal{S}^{ij}_{\text{Obs}}$ and W^{ij}_{LT} is the wedge below $\mathcal{S}^{ij}_{\text{Obs}}$, see Figure 9. To compute W^{ij}_{GT} and W^{ij}_{LT}, the line through the origin and tangent to $\mathcal{S}^{ij}_{\text{Obs}}$ needs to be calculated. For this, we use the derivative of the implicit equation for the ellipse, found in Section 3.2. Note that W^{ij}_{GT} and W^{ij}_{LT} are the tangent lines to the infeasible region $\mathcal{S}^{ij}_{\text{Obs}}$ which is contained within state space $\mathcal{S}^{ij}$, and therefore will always be within the range $[0, 90]$. Both W^{ij}_{GT} and W^{ij}_{LT} can be empty ranges, as shown in Figure 10. If both ranges are empty there is no solution to the problem. If there is no infeasible region, the state space consists of a single wedge W^{ij}_{GT} where

$$W^{ij}_{GT} = [R^{ij}_{IMin}, R^{ij}_{IMax}]$$

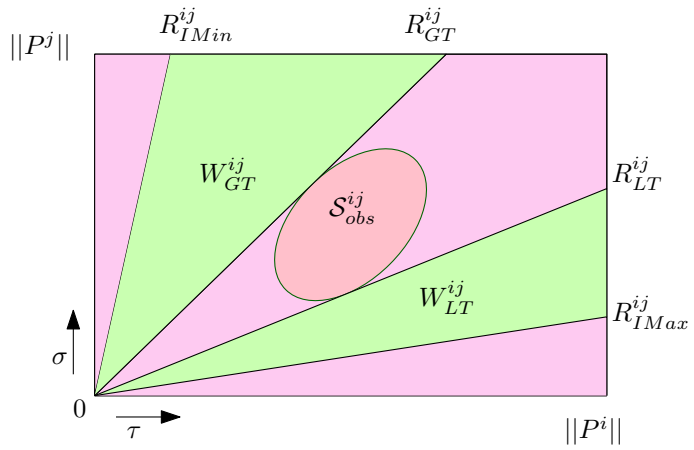

Figure 9: State space with the upper wedge W^{ij}_{GT} and lower wedge W^{ij}_{LT}

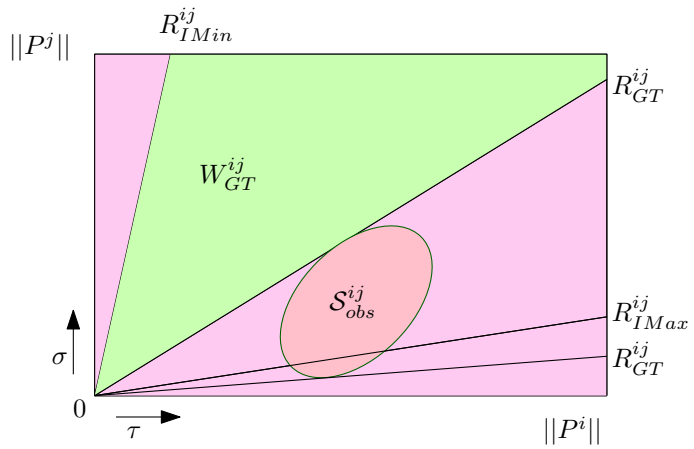

Figure 10: State space with a single wedge W^{ij}_{GT}.

3.5 Visualizing the conflict area

Whenever two bodies have a possibility of not being separated, this can be visually conveyed by showing the *conflict area*. The conflict area A^{ij} consists of two ranges, $[\tau_S, \tau_E], [\sigma_S, \sigma_E]$. The order in which B_i and B_j enter A^{ij} is equal to the order in which they exit A^{ij}. If a state space has two wedges, the choice of within which wedge the angle θ is chosen equals the choice of which body enters A^{ij} first.

To compute A^{ij}, two states need to be calculated, $s_S = (\tau_S, \sigma_S)$ and $s_E = (\tau_E, \sigma_E)$. When B_i and B_j are in state s_S, the only solution for which B_i and B_j stay separated is when either B_i moves at maximum speed and B_j at minimum speed, or when B_i moves at minimum speed and B_j at maximum speed. The infeasible region is contained by a R^{ij}_{IMin},R^{ij}_{IMax} aligned bounding box. State s_S is the lower left corner of this bounding box. State s_E is the upper right corner of the axis-aligned bounding box. See Figure 11.

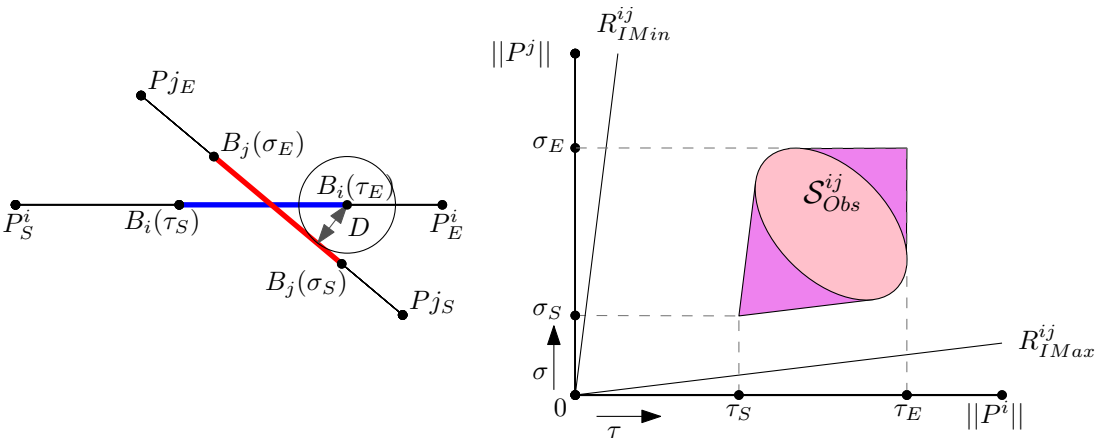

Figure 11: Visualizing the conflict area.

4 From model to linear programming

We will begin by explaining how our problem can be solved by using linear programming, and listing the variables, constraints, and the objective function for our linear solver. Thereafter, we will explain how we convert wedges into branches and stable constraints.

4.1 Forming linear constraints

If for every pair of bodies, the state space has a single wedge available, then feasibility is determined by a set of linear constraints. We therefore choose to use a linear solver to find a solution to our problem. The variables are the *solution speeds*. The solution speed V^i_{Sol} is the speed we seek for body B_i with which B_i remains separated. Logically, V^i_{Sol} is constrained by the minimum and maximum speed V^i_{Min} and V^i_{Max}. The other constraints on V^i_{Sol} originate from the possible ratios from the state space. The possible ratio from state space $\mathcal{S}^{ij}$ concerns two unknowns, V^i_{Sol} and V^j_{Sol}.

For the linear solver, a linear goal has to be set. To reduce the delays in the air space, maximizing the sum of all speeds has been chosen as the objective function. This promotes overall throughput in the air space. The objective function can easily be replaced with any other linear function.

4.2 Branches and stable constraints

To solve the problem, we convert all wedges to linear constraints in order to be able to enter them into a linear solver. Every wedge can be converted to a single linear constraint concerning two unknowns. Because the state space has at most one infeasible region, every wedge has at least one implicit angle defining its range, which do not need to be added to a linear solver.

We differentiate between state spaces with a single wedge and state spaces with two wedges. If there is only one wedge, a *stable constraint* is created representing the range, W^{ij}_{GT}, within which θ can be chosen. If there are two wedges, a *branch* is created representing the ranges W^{ij}_{GT} and W^{ij}_{LT}. Because a linear solver is not capable of handling choices in its constraints, a branch contains a parameter indicating which wedge will be entered in the linear solver. How to explore these branches will be discussed in Section 5.

4.3 Visualizing the solution

To help understand the choices made by the linear solver, we will give an example of how to visualize the solution by showing for which conflict areas the distance between bodies becomes D, and for which conflict areas the bodies would stay separated with a different speed ratio. As discussed in Section 3.1, there can be many different speed ratios in which two bodies with a conflict point stay separated. If we take our previously mentioned goal function into account, maximizing the sum of all speeds, this range is reduced to a single ratio, or two equivalent ratios in cases which are symmetrical. If B_i has no conflict area A^{ij} for any B_j, B_i can assume maximum speed. If B_i

has a conflict area A^{ij}, then either B_i can assume maximum speed or there is a B_j with which B_i will have minimum separation distance D at some point in time.

To visualize the solution for a problem, for each pair of bodies B_i and B_j where A^{ij} is not empty, an indication needs to be given about which body enters A^{ij} first, and which conflict areas limit the speed of B_i. Giving this information can help transform the list of speeds from the solution into a mental image. In Figure 12, this information is conveyed by the red, yellow and blue bridges. The path which gets intersected by the bridge is the path of the body which enters the conflict area second. Red bridges indicate a crossing where the speed of the body gets limited. Blue bridges indicate the order in which the bodies enter the conflict area, but there would have been a possibility to change the ratio. For example, the red bridges crossing P^1 at the intersection of P^1 and P^2 indicates that B_2 will enter the conflict area first, and the speed of B_1 is not its maximum speed because it would otherwise not be separated from B_2.

5 Exploring and reducing the search space

Due to the branching constraints, the linear solver has a vast search space. There is a possibility of a branch for each pair of bodies, resulting in $\binom{n}{2}$ branches. This means there are $2^{\binom{n}{2}}$ possible combinations of constraints for the linear solver in which the optimal solution lies.

In this section, we will start by explaining how to explore the entire search space in a brute force manner. Thereafter, we will show how to reduce this search space by cutting branches without skipping feasible combinations of branches. Then, we will introduce the swap method as an extension to the branch cut. Finally, we mention the four different sortings of the branches with which we will initialize the branch cut.

5.1 Brute force

To iterate over all possible combinations of branches, we view each combination as a binary number. Each branch has a GT and an LT wedge. We will view a 1 as a choice for a GT wedge, and a 0 as a choice for a LT wedge. The first tested combination starts at 0, meaning for every branch h the choice in wedge $W_h = 0$. We will use the binary number as notation, so for example: 1011 has $W_1 = 1$, $W_2 = 0$, etc. As long as the order of the branches remains the same, all possible combinations will be explored when we use all binary numbers. We increment the combination from right to left, as if the least

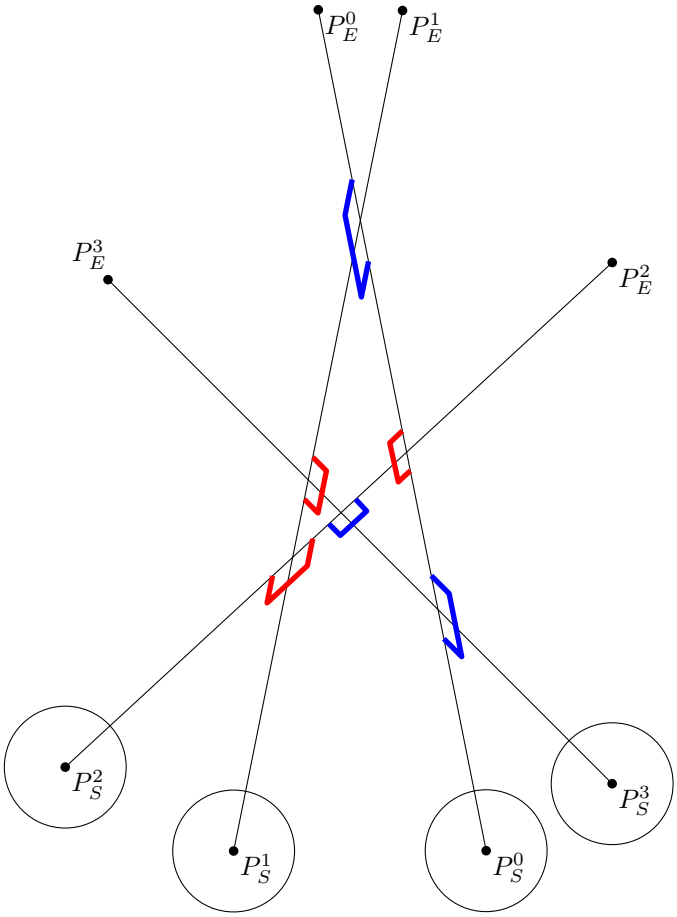

Figure 12: Visualizing the solution.

significant bit is on the right side. Incrementing combination 00111 with 1 will result in combination 01000.

When using brute force exploring, we will enter the combination in the linear solver, run the solver to find a solution, and if the reported solution is better than the previously obtained solution, remember that as our best solution. Then, if not all combinations have yet been tested we increment the combination by 1.

5.2 Branch cut

Branch cut is based on the idea that if a certain combination of a subset of branches, T, has no solution, the total set of branches will also have no solution if it contains T. This means that as soon as such a subset is detected, there is no need to enter additional branches. We start entering the branch combination from left to right. After each addition we check if there is still a feasible region. If a combination starting with 0100 has no solution, we do not have to look at the combinations of branches $5, \ldots, n$. Note that there is a possibility that the first addition of the branch is infeasible, due to the stable constraints. As soon as we encounter an infeasible combination at branch j, we can set every W_i with $i > j$ to 0, and increment the combination from j's position or in other words, incrementing the combination with 2^{n-j}. For example, if combination 010101 is infeasible after adding branch 3, we continue with combination 011000.

To reduce redundant exploration of the search space even further, we will also cut if the best solution for a subset falls below a previously obtained solution. We know that if a subset of combination of branches, T, has a best solution q, the total set of branches will have a best solution of at most q. This means that we can cut the branches as soon as a solution is reported which is worse than the previously obtained solution.

5.3 Swap

We want to be able to cut problematic branches as soon as possible. However, since the we don't know which subset of branches will cause infeasibility, there is a possibility that a branch that causes infeasibility has a high position, making it reduce the search space by only small portion.

The idea is to move branches which cause infeasibility to the front, so that they will be entered first in the linear solver, while still exploring the full feasible search space and without repeating combinations. For every branch we will keep track of how many times the adding of that branch resulted in a branch cut. As soon as we detect a branch that causes such a branch

cut, we add 1 to its counter, and sort all branches based on the number of times they caused a branch cut in descending order from left to right. This reordering needs to be done in a way that no combination is skipped, and no combination is explored multiple times. We enforce these properties by using *HML-sort*, a special kind of insertion sort which only allows branches to be swapped with neighbors that have the same wedge choice W.

To show the workings of HML-sort we will first show that, if we would only explore our search space by incrementing every combination with 1, we would still explore every combination exactly once, even after a reordering at any given moment. Thereafter, we will show how to combine swap sort with branch cut. Let SB be a branch at position i which we will reorder in a combination with n branches. We divide the combination into three parts: high, mid and low. High contains the branches with their position in the range $[0, j-1]$, where $j \leq i$, $W_{j-1} = 1 - W_i$, and $W_m = W_i$ for every m where $j \leq m < i$. Low contains the branches with their position in the range $[i+1, n]$, and mid contains the branches with their position in the range $[j, i]$. See also Figure 13 for a visual representation where $W_i = 0$. After every combination in mid and low has been explored, we look at the next combination of the branches in high, making the order mid and low had previously irrelevant. A reordering of mid will have no influence on which combination of low will be explored after incrementing the entire combination. Therefore, we only need to focus on if incrementing a reordered mid will still explore every combination of its branches exactly once. Consider the two possible wedge choices for W_i. If $W_i = 0$ then, since every wedge choice in mid is 0, we have only explored this one combination of mid and a reordering would still explore every combination of its branches exactly once. If $W_i = 1$ then, since every wedge choice in mid is 1, we have explored all combinations of mid, and can therefore reorder the branches without consequences.

$$\overset{\text{High}}{\left(\begin{matrix} . & . & . & . & . & 1 \end{matrix}\right)} \overset{\text{Mid}}{\begin{matrix} 0 & 0 & \underset{\text{SB}}{0} \end{matrix}} \overset{\text{Low}}{\left(\begin{matrix} . & . & . & . & . & . \end{matrix}\right)}$$

Figure 13: Branch SB with $W_i = 0$.

Now that we know that we can use HML-sort when we only explore our search space by incrementing every combination with 1, we will show how to combine HML-sort with branch cut. Using branch cut, we have to make sure that we do not skip any feasible combinations. Note that it is

impossible to explore a combination multiple times when using branch cut instead of incrementing by 1. Let SB be a branch at position i which causes a branch cut. This means that the subset T containing the branches at position $0, \ldots, i$ has no solution. After the combination is reordered, we need to get the next combination of T. This is achieved by initializing a cut at the highest position of a branch in T. Because we increment T, there is no possibility of skipping feasible combinations. By using HML-sort with branch cut, we possibly sacrifice the cutting of a few branches in the hope of making bigger branch cuts later on.

5.4 Sorting

When exploring the search space with brute force, the initial sorting of the branches is irrelevant. However, when using branch cut, or branch cut with swap, the initial sorting of the branches could have an influence on the runtime. To test the influence of the initial order in which the branches are explored with branch cut and branch cut and swap, we will study four different sortings:

- Random: All branches are put in a random order.
- Incremental: First every branch for body B_i added, then from the remaining branches every branch for B_{i+1}, and so on.
- Smallest wedge : All branches are sorted on the smallest wedge they contain. Branches are added in ascending order of size.
- Sum of wedges: All branches are sorted on the sum of wedges they contain. Branches are added in ascending order of size.

The sortings based on wedge size should quickly limit the feasible region, and thereby accelerate the detection of a possible branch cut.

6 Testing scenarios

In this section we will describe our test setup basics. All tests will be performed with similar bodies. They will have a separation distance D of 5 NM, a minimum speed V_{Min} of 120 knots, and a maximum speed V_{Max} of 250 knots. Because the solution algorithm does not explicitly work with time, the absolute speeds for all bodies could also be chosen as any other set of speeds where $V_{Max} = 2.083 * V_{Min}$. As described in Section 2.2, for each pair of bodies B_i and B_j, the starting points P_S^i and P_S^i will have a minimum

distance of 5 NM from each other. All time measurements will be run on a 6-core Intel i7 4960 with 32 GB RAM, using Gurobi as linear solver.

Test scenario 1 uses a rectangular area with a width of 694.44 NM and a length of 701.39 NM, see Figure 14. Bodies can have a starting point P_S in the southern blue area, which has a width of 694.44 NM and a length of 6.94 NM. The endpoint P_E is can be any position on the northern red line segment.

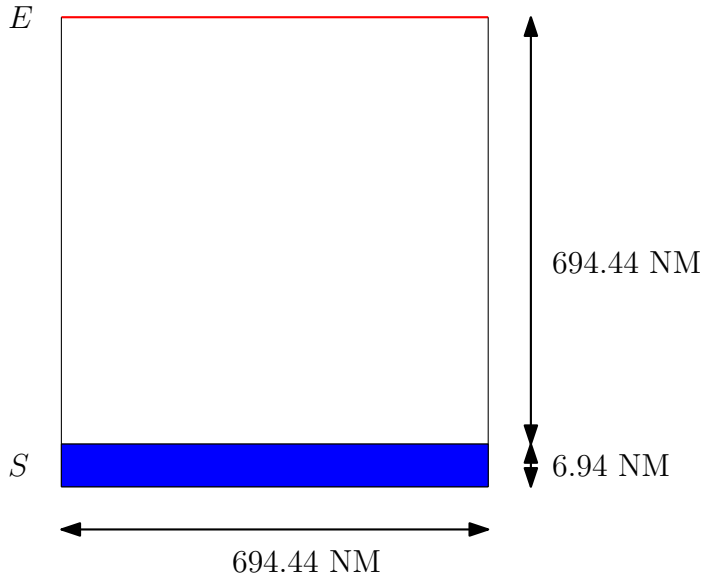

Figure 14: Test scenario 1: Bodies can start in the blue section S and can end on the red line segment E.

Test scenario 2 is an area based on CTA South in the Netherlands with a width of 3854.17 NM and a length of 3652.78 NM, see Figure 15. Bodies will can have a starting point P_S in the light blue areas, marked 1, 2, 4 and 7. The endpoint P_E can be in the orange areas, marked 0, 3, 5, and 6.

7 Tests and results

We will begin this section by listing our test methods and their abbreviations. Thereafter, we will demonstrate the consistency of our time measurements.

We continue by showing that the number of branches has the most influence on the runtime for solving a problem instance, and that the number of bodies and number of stable constraints have minimal influence on the runtime. The relation between bodies and branches in T1 and T2 is examined,

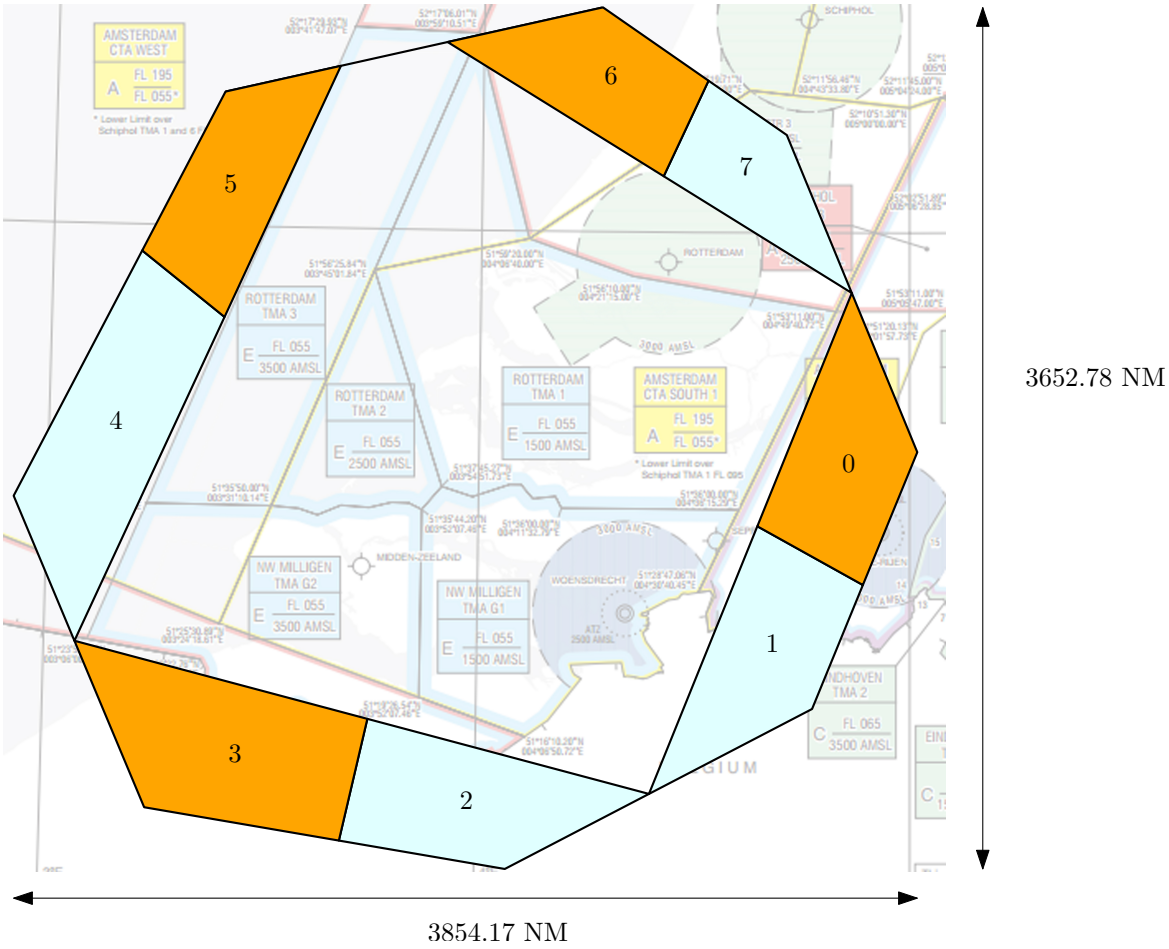

Figure 15: Test scenario 2: Based on CTA South. Bodies can start in the light-blue sections and can end in the orange sections.

and we show that the required runtime for finding a solution is independent of the test scenario by comparing two vastly different test scenarios.

Thereafter we will be comparing runtimes for the different solution methods. We start by comparing the runtime of brute force (BF) with branch cut with random initialization (BC+R). Then, we discuss the influence of the initial sortings for BC and BCS, and thereafter explore the stabilizing effect of the swap method on the sortings. The last runtimes we compare are the fastest branch cut method, branch cut with a sum of wedges sort, with the fastest branch cut and swap method, branch cut and swap with random initialization.

We continue by measuring the runtime of branch cut with sum of wedges sort with up to 222 branches, and discuss the runtime needed for, and occurrence of, unsolvable problem instances. We finish the test section by evaluating the incremental sorting.

7.1 Preliminaries

7.1.1 Tested methods and their abbreviations

We will compare the brute force (BF) method to two other solution methods, branch cut (BC) and branch cut and swap (BCS). Both BC and BCS will be initialized with each of the following four sortings of branches: random (R), incremental (I), wedge (W), and sum of wedges (SW). The initial sorting of the branches for the brute force method is irrelevant. See also Table 1.

Abbreviation	Solution method	Sorting
BF	Brute force	Irrelevant
BC+R	Branch cut	Random
BC+I	Branch cut	Incremental
BC+W	Branch cut	Wedge
BC+SW	Branch cut	Sum of wedges
BCS+R	Branch cut and swap	Random
BCS+I	Branch cut and swap	Incremental
BCS+W	Branch cut and swap	Wedge
BCS+SW	Branch cut and swap	Sum of wedges

Table 1: An overview of the tested solution methods and their abbreviations.

7.1.2 Consistency of time measurements

In the next section we will measure the runtime for solving a problem instance with different solution methods. To be able to compare test results we need to know if the measured runtime is consistent when run multiple times.

In this section we will test the consistency of the runtime for solving a problem instance. For the brute force (BF) solution method we will use a different test setup than the test setup we will use for all the other solution methods. This is because BF takes more time to solve a problem instance than the other solution methods. Using the same setup for BF as for every other solution method would either result in BF taking days to solve a single problem instance, or the other solution methods solving every generated problem instance within a few milliseconds.

Our test setup for BF is as follows. We create 10 random problem instances in T2 for each number of bodies between 10 and 13. With 14 bodies, BF had to run multiple days to solve a single problem instance. Each problem instance is solved 10 times by the BF solution method and its runtime is measured.

Our test setup for all other solution methods is as follows. We create 10 random problem instances in T2 for each number of bodies between 10 and 20. Each problem instance is solved 10 times by the BF solution method and its runtime is measured.

To determine the consistency of a solution method, the average runtime t_{avg} for each problem instance is calculated. Then, for each measured runtime, the ratio r between the measured runtime t_m and the average runtime of that problem instance is calculated by using $r = \max(t_m/t_{avg}, t_{avg}/t_m)$. Looking at Table 2 and Table 3, we can see there are large differences between average runtime and single measured runtime. For example, for one problem instance there was a factor 25.95 difference in runtime between the a single measured runtime and the average runtime with the BC+I solution. Examining the results, this difference in runtime is most likely a result of different startup times for the Gurobi linear solver. If we filter out all problem instances with an average runtime of less than 1000 milliseconds, we see that the solution methods have a much more deterministic runtime, see Table 4 and Table 5. With random initialization of the branches the maximum difference between measured and average runtime is a factor 7.26 for BC, and 2.08 for BCS, which is to be expected from a random initialization. Every other solution method has a median of 1% difference in runtime, and a average difference below 3%. We will therefore choose to only measure the runtime once for each problem instance in the remaining tests.

	BF	BC+R	BC+R	BC+I	BC+S
Min	1,00	1.22	1.12	1.00	1.00
Max	3,18	2.08	10.48	25.95	6.71
Average	1,20	1.57	2.29	1.63	1.22
Median	1,03	1.56	1.77	1.04	1.04

Table 2: Consistency in time measurements for branch cut

	BCS+R	BCS+I	BCS+S	BCS+SW
Min	1.14	1.00	1.00	1.00
Max	7.74	7.41	7.21	8.52
Average	1.67	1.27	1.27	1.36
Median	1.46	1.04	1.04	1.04

Table 3: Consistency in time measurements for branch cut and swap

7.2 Influences on runtime

7.2.1 Most influential variable for runtime

To be able to decide the maximum capacity of the solving methods, we need to know which variables influence the runtime. In this section, we will test the influence on the runtime when varying the number of bodies, the number of stable constraints, and the number of branches. For each variable that we test, we will keep the other two variables constant.

We choose to test problem instances with up to 60 branches. This will give us a broad enough range to see correlation while keeping the runtime manageable. Because we can only generate problem instances based on the number of bodies, we ran tests to determine what the highest number of bodies was which could generate 60 branches or less. As can be seen in Table 6, with 43 bodies or more, all generated problem instances have more than 60 branches. The maximum encountered number of stable constraints in $\mathcal{T}$ is 72. To form our initial test set $\mathcal{T}$ we generated 4000 random problem instances in T2 and stored every unique combination of number of bodies, number of branches, and number of stable constraints.

For each variable, we will test the top two largest subsets in $\mathcal{T}$ where the other two variables are constant. The branch cut with sum of wedges solving method (BC+SW) will be used for the runtime measurement, which we will show to be the fastest solver in Section 7.3.6. From the results we will filter the problem instances without solution. With BC+SW the runtime needed

	BF	BC+R	BC+I	BC+S	BC+SW
Min	1,00	1.12	1.00	1.00	1.00
Max	1,03	7.26	1.14	1.04	1.09
Average	1,01	3.17	1.03	1.01	1.02
Median	1,01	2.81	1.01	1.01	1.01

Table 4: Consistency in time measurements for branch cut, for problem instances with a runtime of 1000 msec or more

	BCS+R	BCS+I	BCS+S	BCS+SW
Min	1.22	1.00	1.00	1.00
Max	2.08	1.02	1.19	1.09
Average	1.57	1.01	1.02	1.02
Median	1.56	1.01	1.01	1.01

Table 5: Consistency in time measurements for branch cut and swap, for problem instances with a runtime of 1000 msec or more

to discover a problem instance has no solution is near constant. This would distort the results. We expand more on the runtime for unsolvable problem instances in Section 7.5.

We start by computing the correlation between the number of bodies and the runtime. The first set we test has 20 branches and 8 stable constraints. With a correlation of 0.56 it indicates a weak positive linear correlation. The second set has 49 branches and 16 stable constraints. With a correlation of 0.11 it indicates there is no linear correlation. Combined with a visual analysis, which can be seen in Figure 16, we conclude there is most likely no linear correlation between number of bodies and runtime.

We continue by computing the correlation between the number of stable constraints and the runtime. The first set we test has 54 branches and 31 bodies. With a correlation of -0.51 it indicates a weak negative linear correlation. The second set has 48 branches and 26 stable constraints. With a correlation of -0.24 it indicates a very weak negative linear correlation. Combined with a visual analysis, which can be seen in Figure 17, we conclude that it is possible that the number of stable constraints has a weak negative linear correlation with the runtime.

Lastly, we compute the correlation between the number of branches and the runtime. The first set we test has 22 bodies and 8 stable constraints. With a correlation of 0.76 it indicates a strong linear correlation. The sec-

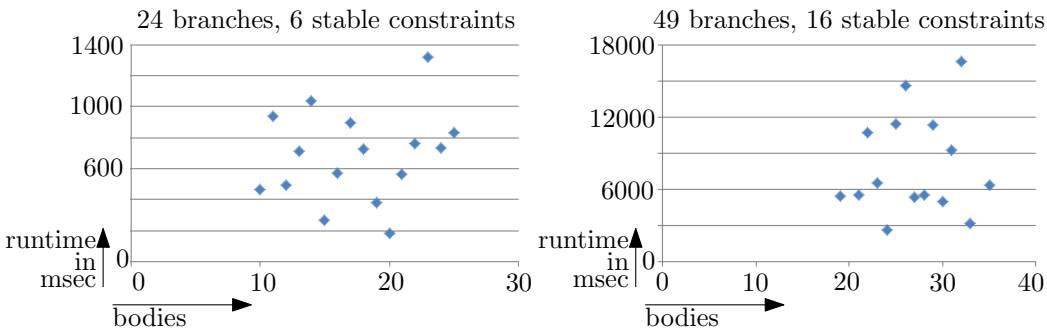

Figure 16: Runtime of the BC+SW when keeping the number of branches and stable constraints constant.

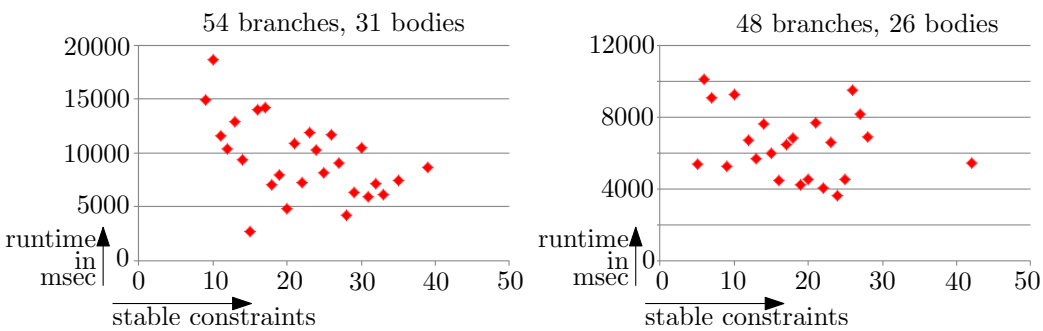

Figure 17: Runtime of the BC+SW when keeping the number of branches and bodies constant.

Bodies	36	37	38	39	40	41	42	43	44	45	46	47	48	49
Branch min	42	46	46	48	40	56	43	68	68	78	71	77	88	97
SC max	57	62	58	65	72	70	71	70	75	85	77	85	85	99

Table 6: Minimum encountered branches and maximum encountered stable constraints for the given number of bodies.

ond set has 48 branches and 26 stable constraints. With a correlation of 0.92 it indicates a very strong linear correlation. Combined with a visual analysis, which can be seen in Figure 18, we conclude that there is a very strong correlation between the number of branches and the runtime. As can be seen in Figure 19, this correlation is most likely exponential. When computing the exponential correlation for both sets, the correlation changes to respectively 0.95 and 0.95, confirming our expectation. A further analysis of the relation between branches and runtime for the BC+SW solver can be found in Section 7.3.6.

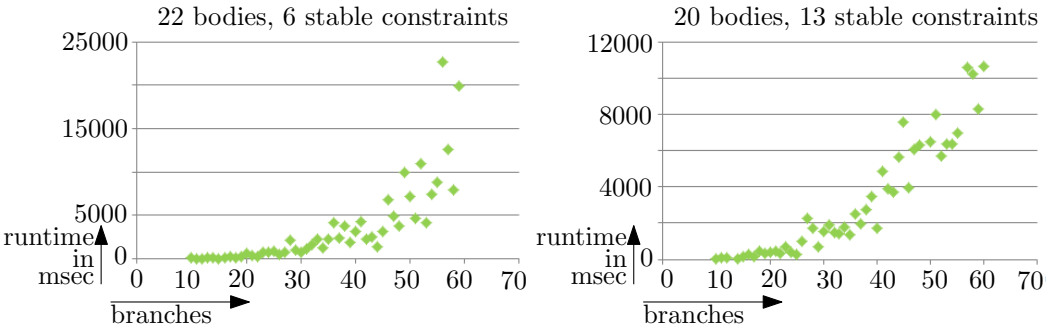

Figure 18: Runtime of the BC+SW when keeping the number of bodies and stable constraints constant.

In summary, the number of branches has the most influence on the runtime. There could be a weak influence from the number of stable constraints, and the number of bodies has most likely no effect on the runtime. We will therefore present the results of the runtime with the number of branches on the horizontal axis.

7.2.2 Branches as a function of bodies in T1 and T2

Since the runtime for solving a problem instance is dependent on the number of branches, but the overall goal is specified in number of bodies within a certain amount of time, it is necessary to examine the relation between the

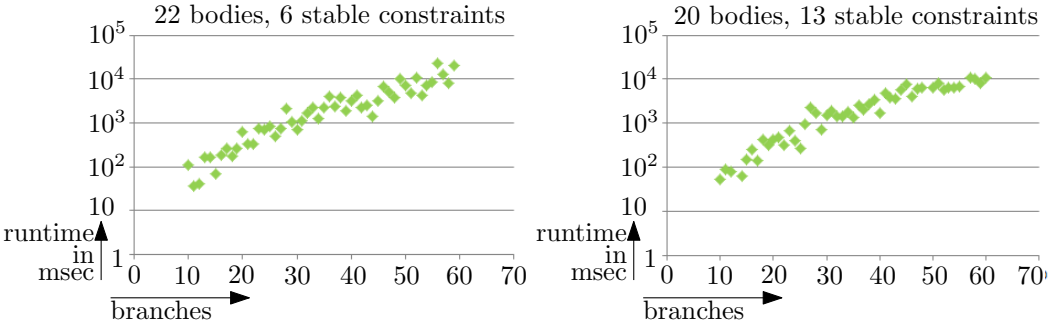

Figure 19: Runtime of the BC+SW when keeping the number of bodies and stable constraints constant on a logarithmic scale.

number of bodies and the number of branches. We create 1000 problem instances in T2 for each number of bodies between 2 and 50 and count the number of branches in each problem instance. In T1 there is a maximum of 11 starting positions. Therefore, we only create problem instances in T1 for each number of bodies between 2 and 11. Figure 20 shows the relation between the number of bodies and the number of branches in T1 and T2. For each number of bodies it shows the percentage of occurrences of a certain number of branches. The percentage of occurrences is colored on a logarithmic scale. Note that the number of branches has an upper bound of $\binom{n}{2}$. As can be seen in Table 7, this theoretical maximum is not reached in T2. T1 has not enough possible starting positions to provide a helpful insight in its true maximum number of branches.

Number of bodies	5	10	15	20	25	30	35	40	45	50
Max possible branches	10	45	105	190	300	435	595	780	990	1225
Max measured branches T1	10	38								
Max measured branches T2	8	28	43	79	105	181	200	242	274	353

Table 7: The maximum possible number of branches compared to the maximum encountered number of branches.

7.2.3 Runtime in T1 versus T2

We compared the runtime of solution methods BC+SW and BCS+R in T1 and T2. As can be seen in Figure 21 and Figure 22, the T1 plot follow the same trend as the T2 plot for both solution methods. Note that for T1, no more than 11 bodies fit in the starting area. We conclude that the runtime is not dependent on test environment.

7.3 Comparing time measurements

To form our test set for the time measurements, we generated 1000 random problem instances in T2 using 20 to 24 bodies and stored every unique combination of number of bodies and number of branches. This resulted in a test set consisting of 6 to 96 branches.

In our comparisons, we remove the problem instances which had no possible solution, this leaves 327 individual problem instances. The reason for removing these problem instances is that the runtime needed to discover a problem instance has no solution is near constant the BC+SW solution method and all solution methods using the swap method. Leaving the unsolvable problem instances in the set would distort the results. We expand more on the runtime for unsolvable problem instances in Section 7.5. Every solving method, except the brute force method, will use this test set for the time measurements. Because brute force (BF) is too slow a solver, we test BF on a smaller set in T2, consisting of 2 to 18 branches.

We will first explain how to read the tables which compare the solution methods. Thereafter, the improvement in runtime branch cut has compared to brute force is shown, using a random initial sorting of the branching constraints. Then, we will demonstrate the influence of the initial sortings on the branch cut solution method. We will then first evaluate the influence of the initial sortings on the branch cut and swap solution method before examining the influence the addition of the swap subroutine has on branch cut. To examine the influence of the swap subroutine, we compare for each initial sorting the runtime for branch cut and branch cut and swap. We finish our time measurements comparisons by comparing the fastest initial sorting of the branch cut solution method to the fastest initial sorting of the branch cut and swap solution method.

7.3.1 Our comparing method

We want to be able to compare the performance of the different solution methods. Because every solution method, except brute force, uses the same

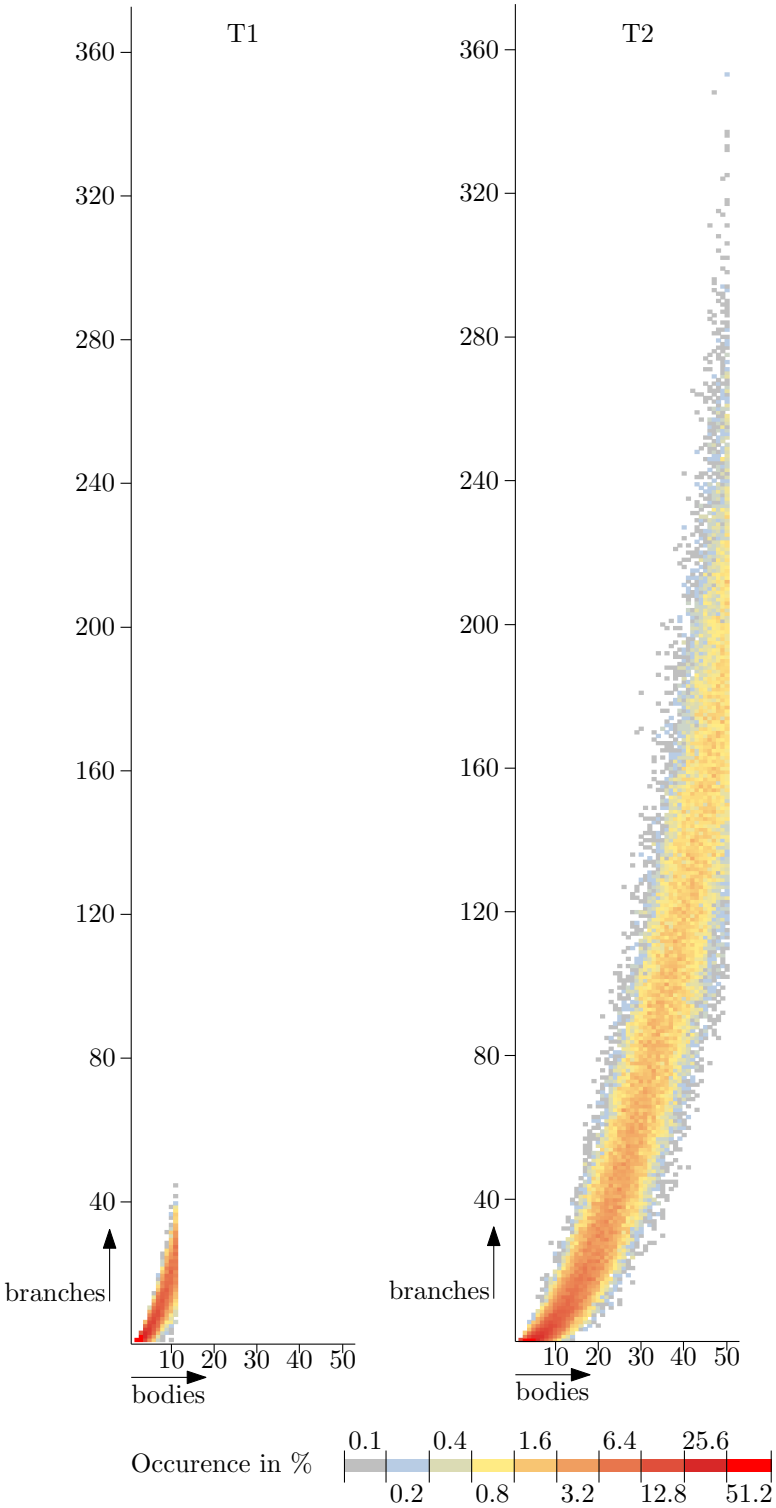

Figure 20: The percentage of occurrences for a certain number of branches when given the number of bodies. Left: Test scenario 1. Right: Test scenario 2

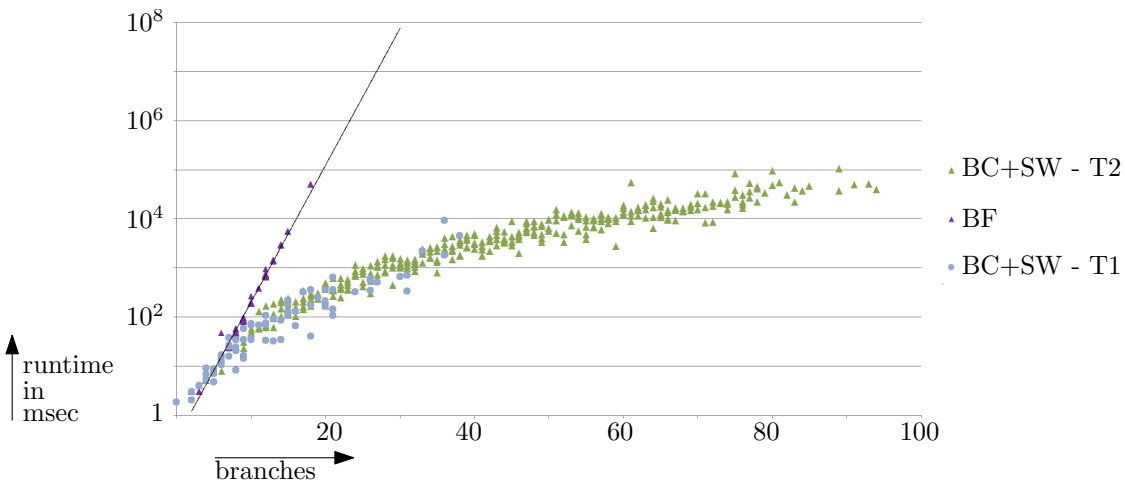

Figure 21: Runtime of branch cut with sum of wedges ordering in T1 (BC+SW - T1) compared to runtime of branch cut with sum of wedges ordering in T2 (BC+SW - T2)

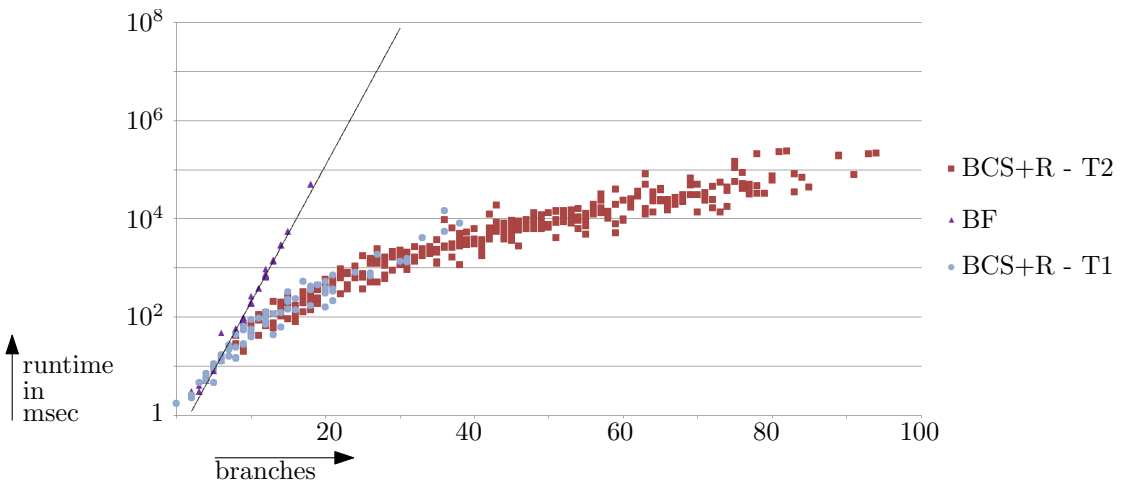

Figure 22: Runtime of branch cut with swap and random ordering in T1 (BCS+R - T1) compared to runtime of branch cut with swap and random ordering in T2 (BCS+R - T2)

set of problem instances, we can compare the runtime per problem instance. Whenever we compare a set solution methods $\mathcal{M}$, we will express the performance on a problem instance for each solution method $m \in \mathcal{M}$ as a linear factor times the minimum time needed for that problem instance by any of the solvers in $\mathcal{M}$. For each solution method we will list the maximum, average, and median performance factor, as well as the percentage of problem instances for which it was the fastest, and the total time in minutes it took to finish solving the entire test set. Note that for a single problem instance there can be multiple solution methods listed as fastest, and the percentages can therefore add up to a number above 100%.

7.3.2 Brute force versus branch cut, random sorting

We compare brute force (BF) to branch cut with random initialization (BC+R), see Figure 23. Both sets depict problem instances in T2. We can see the a great improvement in runtime, especially in the lower bound for solving problem instances, but BC+R has many outliers where the runtime is more than 100 times the lower bound.

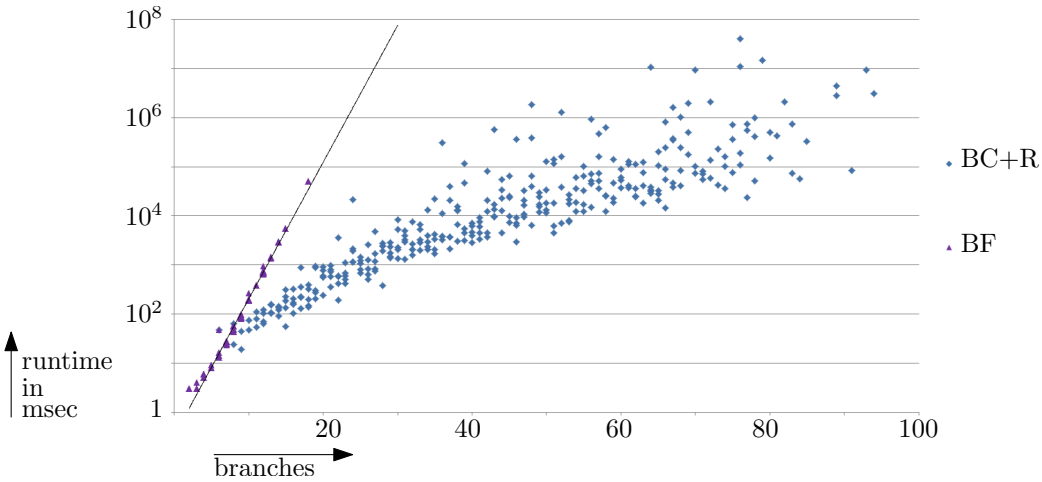

Figure 23: Runtime of brute force (BF) solution compared to branch cut with random initialization (BC+R).

7.3.3 Branch cut, comparing initial sortings

We want to be able to determine the influence of changing the initial sorting of the branching constraints when using the branch cut solution method. Therefore, we will compare branch cut with random initialization (BC+R),

incremental initialization (BC+I), wedge sort initialization (BC+W), and sum of wedges initialization (BC+SW). As can be seen in Table 8, BC+SW is the solution method which takes the least extra time when it's not the fastest, with a maximum of 3.04 times the minimum time any BC solver needed. For 78.59% of the problem instances BC+SW has the lowest runtime and finished solving the full set in 48.62 minutes. We can also see that every sorting finish the test set faster than BC+R. However, BC+I is only the fastest solution method for 4.28% of the problem instances, and has some heavy outliers which are 1934.88 times above the lower bound.

When we visually analyse the four different initial sortings, we can see that every initial sorting has a similar lower bound for solving problem instances, see Figure 24. However, BC+R, BC+I, and BC+W have large outliers which are not observed with BC+SW.

	BC+R	BC+I	BC+W	BC+SW
Max	2070.14	1934.88	744.69	3.04
Average	26.38	22.08	9.25	1.08
Median	2.40	3.08	1.88	1.00
% fastest	9.48	4.28	7.65	78.59
Total time in min	2286.96	1515.58	1906.36	48.62

Table 8: Branch cut initial sortings compared.

7.3.4 Branch cut and swap, comparing initial sortings

We want to be able to determine the influence of changing the initial sorting of the branching constraints when using the branch cut and swap solution method. Therefore, we will compare branch cut and swap with random initialization (BCS+R), incremental initialization (BCS+I), wedge sort initialization (BCS+W), and sum of wedges initialization (BCS+SW). As can be seen in Table 9, BCS+R is the solution method which takes the least extra time when it's not the fastest, with a maximum of 3.40 times the minimum time of any BCS solver needed, but the difference in solvers is not as obvious as with the BC solution, For 34.86% of the problem instances BCS+R has the lowest runtime. The total runtime of BC+R, BC+I, and BC+W are very similar, with only BCS+SW taking 1.54 times longer to finish the test set compared to BCS+R.

When we visually analyse the four different initial sortings, we can see that every initial sorting has a similar lower bound for solving problem instances,

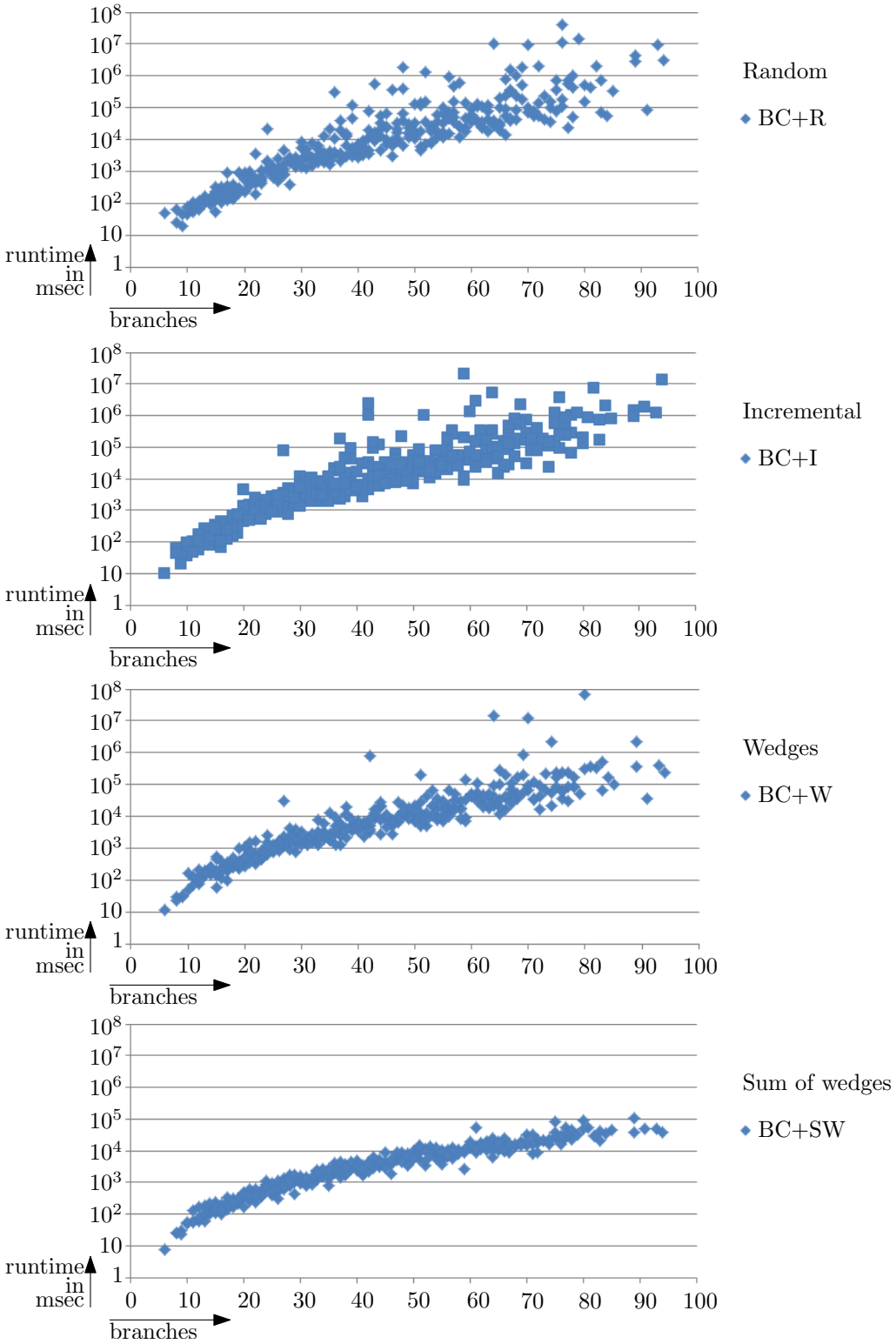

Figure 24: Runtime of branch cut with all four different initial sortings.

see Figure 25. There are some small outliers noticeable with BCS+SW around the 90 branches, which could explain why its total time 1.54 times higher than BC+R.

	BCS+R	BCS+I	BCS+W	BCS+SW
Max	3.40	5.53	5.75	13.33
Average	1.26	1.48	1.41	1.54
Median	1.10	1.31	1.26	1.33
% fastest	34.86	19.88	25.08	21.10
Total time in min	90.35	107.78	100.17	138.95

Table 9: Branch cut and swap initial sortings compared.

7.3.5 Branch cut versus branch cut and swap, same sorting

To study the effect of adding the swap subroutine to the branch cut solution method, we do a pairwise comparison. We start by comparing branch cut with random initialization (BC+R) to branch cut and swap with random initialization (BCS+R). When putting Figure 24 and Figure 25 side by side, it can be observed that the lower bound for solving a problem instance is the same for BC+R as for BCS+R, but the outliers that occurred with BC+R do not exist when adding the swap routine to the solution method.

Table 10 shows that, although BCS+R finished the entire set considerably faster than BC+R, the addition of the swap method only removes the outliers and does not greatly reduce the lower bound for solving a problem instance. This can be seen by the median for BC+R, which is 1.85, and the fact that in 20.18% of the problem instances BC+R was the fastest.

	BC+R	BCS+R
Max	799.34	2.99
Average	14.86	1.07
Median	1.85	1.00
% fastest	20.18	79.82
Total time in min	2286.96	90.35

Table 10: BC+R compared to BCS+R.

When comparing the time measurements for branch cut and branch cut and swap it can be observed that the addition of the swap method also removes the outliers which occured with the incremental and smallest wedge

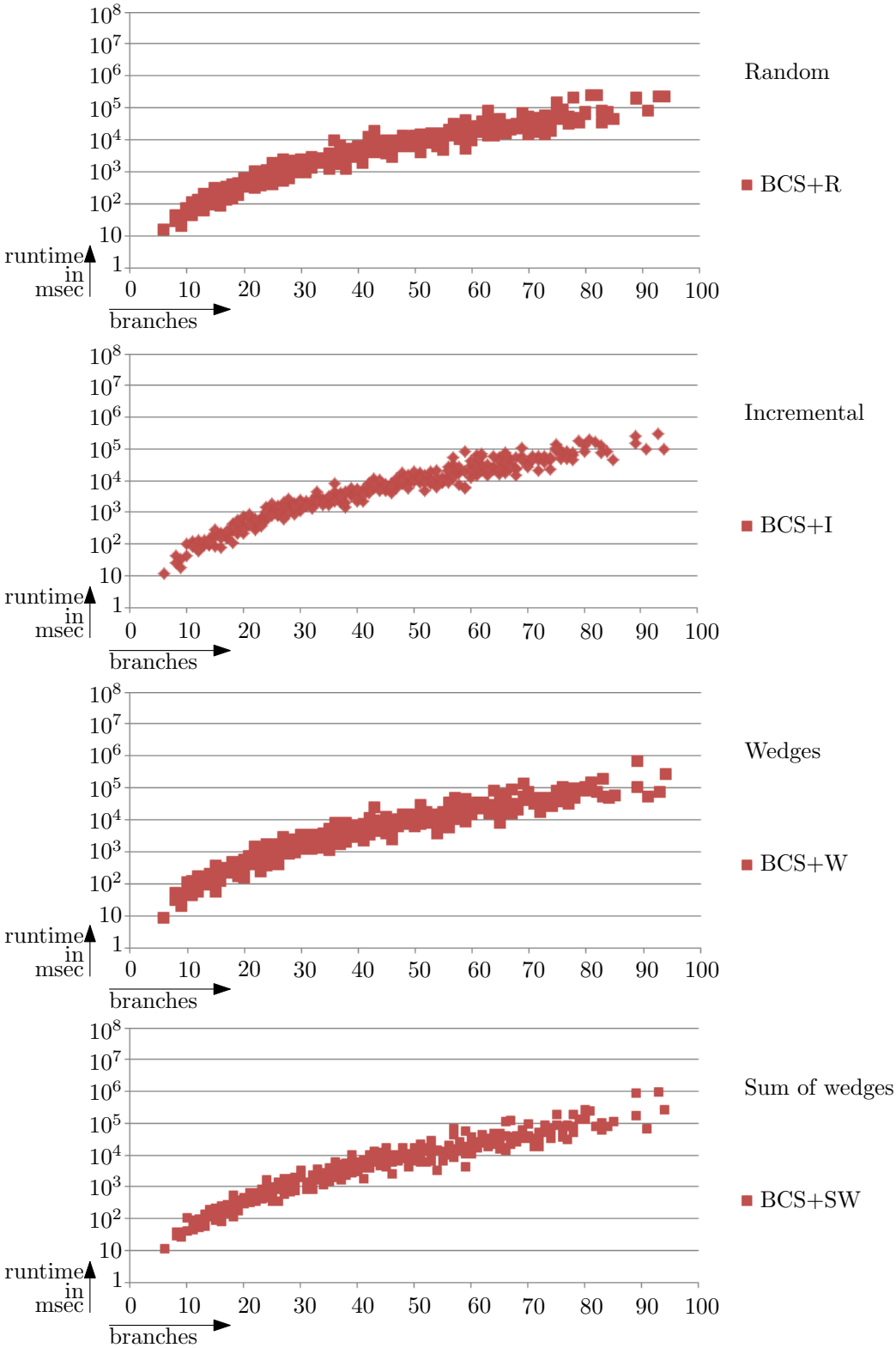

Figure 25: Runtime of branch cut and swap with all four different initial sortings.

initialization. The addition of the swap subroutine results in a faster solution method for 84.40% of the problem instances with the incremental initialization, and for 72.48% of the problem instances with the smallest wedge initialization. See also Table 11 and Table 12. However, BC+SW has no

	BC+I	BCS+I
Max	1086.17	2.02
Average	10.76	1.03
Median	1.93	1.00
% fastest	15.60	84.40
Total time in min	1515.58	107.78

Table 11: BC+I compared to BCS+I.

	BC+W	BCS+W
Max	671.78	3.15
Average	6.67	1.11
Median	1.34	1.00
% fastest	27.83	72.48
Total time in min	1906.36	100.17

Table 12: BC+W compared to BCS+W.

outliers, and adding the swap subroutine to this solver slows the solver down. As can be seen in Table 13, BC+SW is faster than BCS+SW for 81.95% of the problem instances. BCS+SW takes 2.86 times as long to complete the full set of problem instances, and is a maximum of 19.09 times as slow on specific problem instances.

	BC+SW	BCS+SW
Max	2.17	19.09
Average	1.05	1.90
Median	1.00	1.53
% fastest	81.96	18.04
Total time in min	48.62	138.95

Table 13: BC+SW compared to BCS+SW.

7.3.6 Fastest of branch cut versus fastest of branch cut and swap

We compare the overall fastest branch cut solver, BC+SW, to the overall fastest branch cut and swap solver, BCS+R. The measured runtimes for each solver are combined in Figure 26.

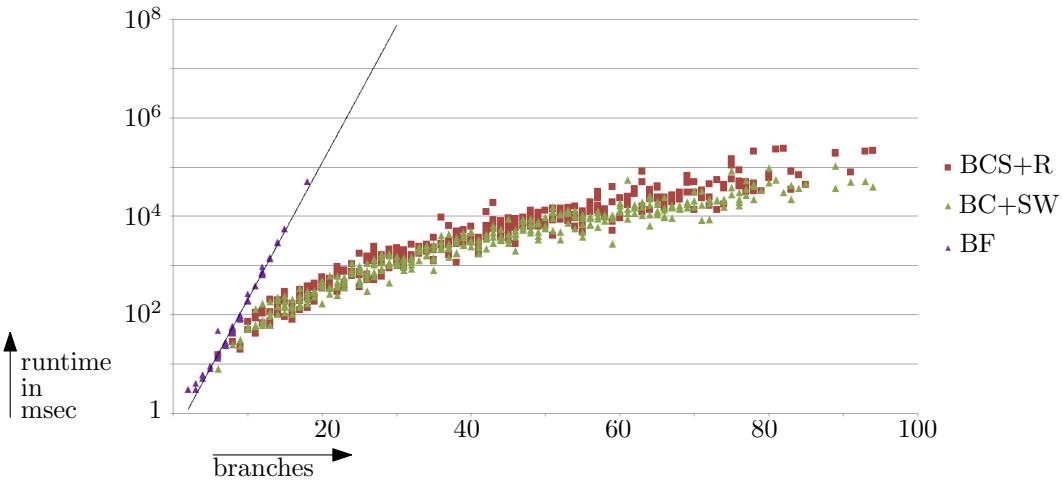

Figure 26: Runtime of branch cut and swap with random initialization (BCS+R) compared to branch cut with sum of wedges initialization (BC+SW).

We can see that the runtime for solving with BC+SW is slightly lower for equal problem instances than for solving with BCS+R. When comparing BC+SW and BCS+RW, the lowest runtime has been selected and for each solution method the runtime has been expressed as a factor of the lowest time, see also Table 14. Solving with BC+SW results in the lowest runtime in 72.17% of the tested problem instances. BC+SW also had the lowest maximum factor it took extra compared to the lowest time, the best average factor, and the lowest median factor.

	BC+SW	BCS+R
Max	2.42	7.77
Average	1.07	1.58
Median	1.00	1.30
% fastest	72.17	28.13

Table 14: Comparing performance

7.4 Performance of BC+SW

The goal of our algorithm is to be able to find a solution for 15 bodies within 5 minutes. Configurations with up to 232 branches have been measured for runtime, see Figure 27. When allowing a maximum of 5 minutes runtime, 107 branches can be handled. This equals a maximum of 25 conflicting airplanes in T2, see Figure 20 and accompanying Table 7. From Table 7, we also can see that a random generated problem instance with 15 bodies has up to 43 branches. The highest measured runtime for solving a problem instance with 43 branches with the BC+SW solver is 6742 msec, well within our targeted 5-minute bound.

To see the trend of the runtime for BC+SW, we use Excel to plot the trendline. The best fitting function is $y = 0.0049x^{3.6069}$ with a coefficient of determination R^2 of $R^2 = 0.9478$, meaning that the function fits our results very well.

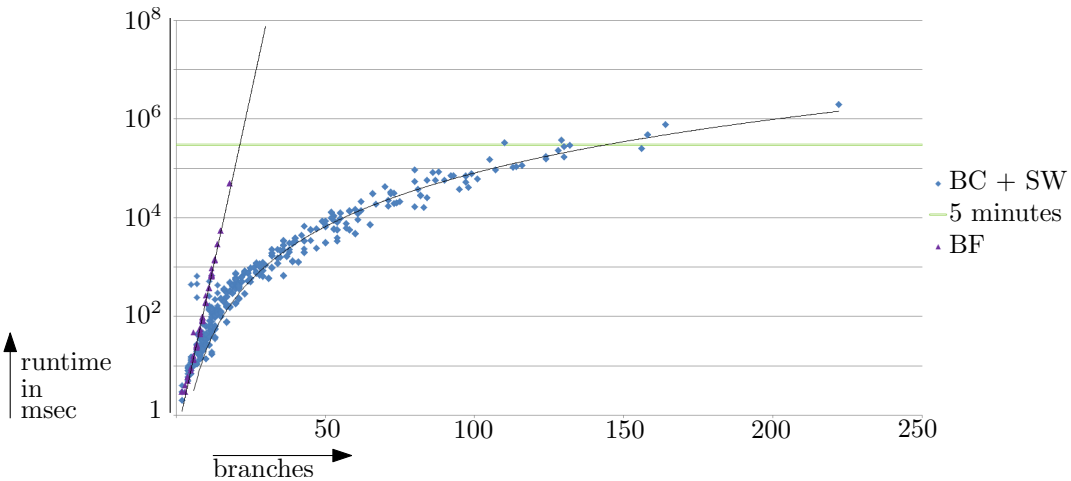

Figure 27: Runtime of branch cut with sum of wedges ordering (BC+SW) with up to 250 branches.

7.5 Unsolvable situations

Not every problem instance can be solved. Out of the 330 problem instances tested in Section 7.3, only 3 gave unsolvable situations. With BC+SW, these situations are detected in less than 11 milliseconds, regardless of number of branches. In comparison, the BC+I solution method took 9.24 hours to solve one of these unsolvable problem instances.

The near constant runtime for detecting these unsolvable problem instances is most likely due to the existence of an empty branch in the problem instance, which gets evaluated first, resulting in a direct detection of the unsolvability.

7.6 Evaluation incremental ordering

As can be seen in Section 7.3.3 and Section 7.3.4, the incremental order only slightly reduces the runtime with BC. This can be explained by looking at one of the tested problem instances. In Figure 28 an example of branches is visualized as a graph where every body B_i is represented by node i, and every branch for B_i and B_j is represented by an edge between i and j. The idea of the incremental order is to first explore the branches that influence each other. Because not every pair of bodies causes a branch, and the numbering of the bodies has no relation to their relative position in the graph, the incremental order based on numbering does not work.

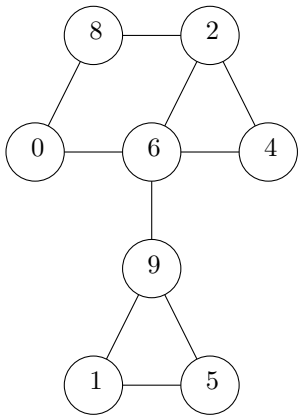

Figure 28: Graph representing the branches with bodies as nodes and branches as edges.

In order for the idea of incremental order to work, the adding of the branches needs to be graph based, instead of numerical based.

8 Conclusion and future work

This thesis was set out to form a decision support system that could keep airplanes separated by only modifying their speed. Based on real life data, the goal was to be able assign a speed to 15 airplanes within 5 minutes. We

build a model that simplifies airplanes to bodies, and performs far above the previously set requirement. It can handle even the most conflicting problem instances of 15 bodies in less than 7 seconds, and can solve up to 25 bodies within 5 minutes. We found that the runtime depends on the number of branches in the problem instance, or in other words the number of conflicting bodies for which a choice can be made which body has priority at their intersection. The program calculates the number of branches before starting to solve, and can therefore indicate if the problem instance is too hard to solve. A useful addition, for the use in air traffic control, would be to allow for indication of priorities by the air traffic controller. This would reduce branches, allowing for even more bodies to be handled by the solver. In normal situations, the number of branches will be low enough to solve within seconds. The model we used is quite simplified. It would be interesting to look into the influence on runtime when adding more realism, such as wind, multiple flight levels and air fuel optimizers like Vela et al. [14].

A secondary goal was to be able to convey the choices made by the program visually. We showed that the information from our decision support system can be visually conveyed similar to the PHARE [9] system. By showing before which point the priority of aircrafts on their intersection can still be changed, and visualizing the solution of our solver by showing the which intersections limit the throughput of the aircrafts, the mental image of an air traffic controller can be maintained. However, with our current model it is not useful to further test the clarity of such a visualization.

For solving our problem, we used three different solution methods for exploring our search space: a brute force method, a branch cut method which pruned the binary tree while exploring the search space, and a branch cut with swap method which added dynamic reordering of the branches to the branch cut solution method. The brute force solution method was significantly slower than the methods using branch cut, taking days instead of minutes to solve problem instances.

With our branch cut solution methods, we tested the influence of the initial order of the branches in the tree on the runtime. We compared a random initialization of the branches to three different initial sortings: incremental, smallest wedge size, and sum of wedge size. When using the branch cut with swap solution method the random order of branches in the tree was the fastest most of time although the other sortings were not much slower, with exception of the sum of wedge size sorting. This could be because a random initialization allowed for the branches to be freely rearranged at the beginning of exploring the search space, without getting stuck in a local optimum. The reason why the sum of wedges performed the worst could be be because the sorting is already near optimal, which gives false positives on

which branch causes a branch cut to occur.

When looking at the branch cut solution method, it was the random sorting which was the slowest initialization and the sum of wedges sorting showed a huge improvement. It was on average 47 times faster than a random initialization and 1.86 times faster than the branch cut with swap and random initialization solution method. It was to be expected that the sum of wedges would be the fastest initial ordering, because a small sum of wedges limits the search space with both choices of wedges, and therefore would quickly cause a branch cut to occur. The smallest wedge sorting leaves the possibility for a large wedge to be at the top of the tree, and thereby not limiting the search space fast enough. The slow performance of the random sorting further suggest that infeasibility is caused by subsets of branches. If some of them are low in the decision tree, the possible branch cuts are small, and thereby leaving a large search space to explore.

Even though we found a solution method without the swap method to be the fastest, the swap method can still be of use. The branch cut solution method showed many outliers, where the runtime is more than 100 times more than the trend would indicate, for all sortings except the sum of wedge size sorting. Both the addition of the swap method and the changing of the initial sorting removed these outliers. The lower bound for the runtime remained the same. This leads us to believe that only when a good initial sorting is known for the constraints, the addition of the swap method is slowing the program down. However, if no good initial sorting of constraints is known, the swap method stabilizes the branch cut. It should be tested if the swap method would show the same behaviour on other branching linear programming problems.

The incremental ordering we tested was flawed. It would be interesting to see if the incremental order based on which clusters there are in the graph with branches could improve results even further. Such a graph based approach to adding branches could even continue in exploring ways to divide the branches in multiple disjoint subsets to reduce time.

References

[1] Averty, P., Johansson, B., Wise, J., & Capsie, C. 2007. Could Erasmus Speed Adjustments Be Identifiable By Air Traffic Controllers?. *7th USA/Europe Air Traffic Management Research and Development Seminar*

[2] Cafieri, S., & Durand, N. 2013. Aircraft deconfliction with speed regulation: new models from mixed-integer optimization. *Journal of Global Optimization*, Volume 58, Issue 4, 613-629.

[3] Eby, M. S. 1995. A self-organizational approach for resolving air traffic conflicts. *Lincoln Lab. J.* 7, 2 (September 1995), 239-254.

[4] Ehrmanntraut, R. 2004. The potential of speed control. *Proc. 23rd Digital Avionics Systems Conference (DASC 2004)*, vol. 1, Oct. 2004, 3.E.33.17.

[5] Guy, S.J., van den Berg, J., Lin, M.C., & Manocha, D. 2010. Geometric Methods for Multi-Agent Collision Avoidance *Proceedings of the twenty-sixth annual symposium on Computational geometry*, 115-116.

[6] ICAO. 2007. *Procedures for Air Navigation Services Air Traffic Management (PANS-ATM, Doc 4444 ATM/501)*, Fifteenth Edition.

[7] Idris, H. 1994. Human-Centered Automation of Air Traffic Control Operations in the Terminal Area Flight Transportation Laboratory. MIT.

[8] Jones, J.C., Lovell, D.J., & Ball, M.O. 2013. En Route Speed Control Methods for Transferring Terminal Delay. *Proceedings of the Tenth USA/Europe Air Traffic Management Research and Development Seminar (ATM2013)*, Chicago, IL, USA.

[9] Jorna, P.G.A.M., Pavet, D., van Blanken, M., & Pichancourt, I. 1999. PHARE Ground Human Machine Interface (GHMI) project: Summary report. EUROCONTROL.

[10] LaValle, S. M. 2006. Planning Algorithms. Cambridge University Press, Cambridge, UK, 2006.

[11] Performance Review Commission. 2013. Performance review report: An assesment of air traffic management in europe during the calendar year 2013. Eurocontrol, Brussels, Belgium, Tech. Rep., March 2014.

[12] SESAR factsheet. 2014. SESAR 2020: developing the next generation of European Air Traffic Management. Press-release by ec.europa.eu, European Union, retrieved from http://ec.europa.eu/research/press/jti/factsheet_sesar-web.pdf on 22-02-2015.

[13] Steria, G.G., Allignol, C., & Durand, N. 2001. The Influence of Uncertainties on Traffic Control using Speed Adjustments. *9th USA/Europe Air Traffic Management Research and Development Seminar*

[14] Vela, A., Solak, S., Singhose, W., & Clarke, J.P. 2009. A Mixed Integer Program for Flight-Level Assignment and Speed Control for Conflict Resolution. *Joint 48th IEEE Conference on Decision and Control and 28th Chinese Control Conference*, 5219 - 5226.

A Formula rearrangements from Section 3.2

Rearranging equation 1:

$$\begin{aligned}(\sigma - \tau\cos(\alpha))^2 + (0 - \tau\sin(\alpha))^2 &= D^2\\ \sigma^2 - 2\cos(\alpha)\sigma\tau + \tau^2\cos(\alpha)^2 + \tau^2\sin(\alpha)^2 &= D^2\\ \sigma^2 - 2\cos(\alpha)\sigma\tau + \tau^2 - D^2 &= 0 \qquad (5)\end{aligned}$$

Showing that if $A = C$, $a^2 = b^2$ or $\varphi = 45$:

$$\begin{aligned}A &= C\\ a^2(\sin\varphi)^2 + b^2(\cos\varphi)^2 &= a^2(\cos\varphi)^2 + b^2(\sin\varphi)^2\\ a^2((\sin\varphi)^2 - (\cos\varphi)^2) &= b^2((\sin\varphi)^2 - (\cos\varphi)^2)\\ a^2 = b^2 \vee (\sin\varphi)^2 - (\cos\varphi)^2 &= 0\\ a^2 = b^2 \vee \varphi &= 45\end{aligned}$$

Rearranging to get equations 3 and 4, part 1:

$$\begin{aligned}\frac{B}{A} &= -2\cos(\alpha)\\ \frac{b^2 - a^2}{\frac{1}{2}a^2 + \frac{1}{2}b^2} &= -2\cos(\alpha)\\ b^2 - a^2 &= -\cos(\alpha)a^2 - \cos(\alpha)b^2\\ b^2 + \cos(\alpha)b^2 &= a^2 - \cos(\alpha)a^2\\ a^2 &= \frac{b^2 + \cos(\alpha)b^2}{1 - \cos(\alpha)} \qquad (6)\\ b^2 &= \frac{a^2 - \cos(\alpha)a^2}{1 + cos(\alpha)} \qquad (7)\end{aligned}$$

Rearranging to get equations 3 and 4, part 2:

$$\begin{aligned}\frac{F}{A} &= -D^2\\ \frac{-b^2a^2}{\frac{1}{2}a^2 + \frac{1}{2}b^2} &= -D^2\\ -b^2a^2 &= -\frac{1}{2}D^2b^2 - \frac{1}{2}D^2a^2 \qquad (8)\end{aligned}$$

Substituting a^2 for equation 6 on the right hand side of equation 8 to get equation 3:

$$
\begin{aligned}
-b^2a^2 &= -\frac{1}{2}D^2b^2 - \frac{1}{2}D^2\frac{b^2+\cos(\alpha)b^2}{1-\cos(\alpha)} \\
a^2 &= \frac{1}{2}D^2 + \frac{1}{2}D^2\frac{1+\cos(\alpha)}{1-\cos(\alpha)} \\
a^2 &= \frac{\frac{1}{2}D^2 - \frac{1}{2}D^2\cos(\alpha) + \frac{1}{2}D^2 + \frac{1}{2}D^2\cos(\alpha)}{1-\cos(\alpha)} \\
a^2 &= \frac{\frac{1}{2}D^2}{1-\cos(\alpha)}
\end{aligned}
$$

Substituting b^2 for equation 7 on the right hand side of equation 8 to get equation 4:

$$
\begin{aligned}
-b^2a^2 &= -\frac{1}{2}D^2\frac{a^2-\cos(\alpha)a^2}{1+\cos(\alpha)} - \frac{1}{2}D^2a^2 \\
b^2 &= \frac{1}{2}D^2\frac{1-\cos(\alpha)}{1+\cos(\alpha)} + \frac{1}{2}D^2 \\
b^2 &= \frac{\frac{1}{2}D^2 - \frac{1}{2}D^2\cos(\alpha) + \frac{1}{2}D^2 + \frac{1}{2}D^2\cos(\alpha)}{1+\cos(\alpha)} \\
b^2 &= \frac{\frac{1}{2}D^2}{1+\cos(\alpha)}
\end{aligned}
$$