VectorRace
Finding the Fastest Path Through a Two-Dimensional Track

Jakob Schmid
schmid@cs.aau.dk

31st May 2005

Abstract

The algorithm described herein finds the fastest path of an accelerating vehicle through an arbitrary two-dimensional track. The algorithm is usable for finding paths for virtual or real computer-controlled accelerating vehicles. The abstraction for this task is the game of VectorRace, which is formally defined in the article. The algorithm operates on discrete values, and the track is defined as line segments between whole-numbered points. The article contains a time and space complexity analysis of the algorithm. Furthermore, a solution to the problem using binary decision diagrams are investigated, but it is argued that the aformentioned algorithm is more efficient.

1 Introduction

Problems of finding the shortest path through mathematically defined environments have been the subject of much study, and methods such as Dijkstra's algorithm have been devised to solve such problems[4].

The problem discussed in this article is related to the graph theoretical *shortest path problem*, but is different in that the paths investigated in this article are subject of acceleration rules.

The problem is modelled by the paper-and-pencil game called VectorRace. VectorRace is a turn-based game, where a number of players must maneuver accelerating vehicles represented by line segments through a track from a starting point to a goal line. In this article, the game is reduced to have only a single player. Figure 1 on the following page shows a sample single-player game of VectorRace.

The VectorRace model developed in this article is *discrete* in the sense that all points in the model are whole numbers, except for *collision points*, which are defined by the intersection between two line segments. These collision points have rational coordinates which are often not whole numbers.

The intuitive problem definition is:

> Given a VectorRace track, find the path from the starting point to the goal line with the fewest number of line segments.

A more formal problem definition is developed later in the article.

Notation

In this article, the following conventions are used:

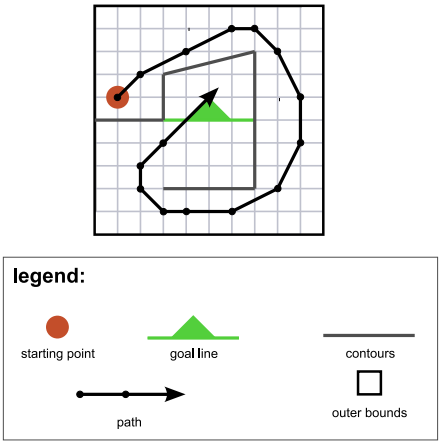


Figure 1: A sample VectorRace game

- Vectors are written in boldface, e.g. $\mathbf{v}$, $\mathbf{r}$, etc. The elements of a 2-dimensional vector $\mathbf{v}$ are denoted v_x and v_y, respectively.
- Points are written as uppercase letters, e.g. A, B, etc. The coordinates of a 2-dimensional point A are denoted A_x and A_y, respectively.
- Sets are generally written in this format: Set, except the set of whole numbers, $\mathbb{Z}$. Sets superscripted with a number n denote the cartesian product of n sets, e.g. $\mathbb{Z}^2 = \mathbb{Z} \times \mathbb{Z}$.

2 VectorRace Model

The VectorRace game is modelled by a *track* definition and a *game* definition. The track definition depends upon a definition of *line segments*.

2.1 Line Segment

A *line segment* is defined as a pair:

$$\langle A, \mathbf{r} \rangle, \qquad A, \mathbf{r} \in \mathbb{Z}^2$$

The set of points (x, y) in the line segment satisfy the equation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = A + t\ \mathbf{r}, \qquad t \in [0\,;1]$$

Since the line segments have a starting point A and an ending point $A + \mathbf{r}$, they are actually *directed* line segments. However, for the sake of brevity, we shall just call them line segments.

To investigate whether two line segments $\langle A, \mathbf{r} \rangle$ and $\langle B, \mathbf{s} \rangle$ intersect, we solve:

$$A + t\ \mathbf{r} = B + u\ \mathbf{s}$$

The solutions are[1]

$$(t, u) = \left[\frac{\begin{vmatrix} (B_x - A_x) & -s_x \\ (B_y - A_y) & -s_y \end{vmatrix}}{\begin{vmatrix} r_x & -s_x \\ r_y & -s_y \end{vmatrix}}, \frac{\begin{vmatrix} r_x & (B_x - A_x) \\ r_y & (B_y - A_y) \end{vmatrix}}{\begin{vmatrix} r_x & -s_x \\ r_y & -s_y \end{vmatrix}} \right]$$

If $t, u \in [0\,;1]$, then the two line segments intersect.

2.2 Track

2.2.1 Intuition

A VectorRace *track* is modelled as a 6-tuple consisting of two points defining a rectangle that is the *outer bounds* of the track, a set of line segments defining the *contour* of the track, a *starting point*, and a *goal line* with an accompanying *crossing direction vector*, specifying the correct direction in which to cross the goal line.

[1] Recall that $\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1$.

2.2.2 Definition

A Vectorrace *track* is a 6-tuple, $\langle BL, TR, \text{Con}, S, GL, \mathbf{CD} \rangle$.

- $BL, TR \in \mathbb{Z}^2$ are two points defining the *outer bounds* of the track. We say that a point A is *within bounds* if

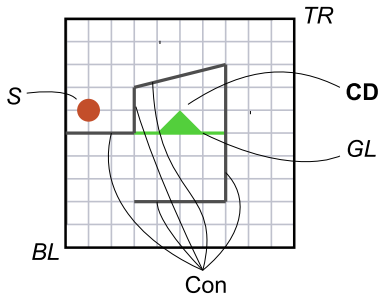
$$BL_x < A_x < TR_x \quad \wedge \quad BL_y < A_y < TR_y$$

  We say that a point A is *on the bounds* if

$$\begin{aligned} & A_x = BL_x \vee A_x = TR_x \\ \wedge \quad & A_y = BL_y \vee A_y = TR_y \end{aligned}$$

- Con is a finite set of line segments defining the *contours* of the track. For all $\langle A, \mathbf{r} \rangle \in$ Con, A and $A + \mathbf{r}$ must be within bounds or on the bounds.
- $S \in \mathbb{Z}^2$ is the *starting point* of the track. S must be within bounds.
- GL is a line segment defining the *goal line*. If $GL = \langle A, \mathbf{r} \rangle$, A and $A+\mathbf{r}$ must be within bounds or on the bounds.
- $\mathbf{CD} \in \mathbb{Z}^2$ is the *crossing direction vector*, which must fulfil $\mathbf{CD} \perp GL$

Figure 2 shows an example of a very simple track.



(Note that **CD** is shown as a wide arrow on top of the goal line pointing in the direction of **CD**.)

Figure 2: Example track

2.3 Game

2.3.1 Intuition

A VectorRace game is modelled as a sequence of *configurations* selected by the player, each configuration being a pair $\langle A, \mathbf{v} \rangle$, where the point A determines the position of the player's vehicle and the vector $\mathbf{v}$ defines the current velocity of the vehicle.

In the first turn, the starting point A_0 is set to the starting point of the track, S. The starting velocity is always $v_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. In the following turns, the player may optionally modify the velocity vector of the previous move with 1 in any direction. This is called *acceleration*, and is denoted $\mathbf{a}$. This notion of acceleration yields a maximum of 9 different configurations each turn.

The game is finished in one of two cases:

1. The game is *won* if the vehicle crosses the goal line in the direction specified by **CD**
2. The game is *lost* if:
   (a) the line segment from the position of the previous configuration to the position of the current configuration intersects with a line segment of the track
   (b) the current configuration is out of bounds

2.3.2 Definition

In the following, $\{-1, 0, 1\}^2$ denotes the set of all vectors where both coordinates are elements in the set $\{-1, 0, 1\}$.

A *configuration* is defined as a pair $\langle A, \mathbf{v} \rangle$, where A is the position of the vehicle, and $\mathbf{v}$ is the velocity.

Given a track $\langle BL, TR, \mathsf{Con}, S, GL, \mathbf{CD} \rangle$, we define a *game* to be a finite sequence of configurations $\{\langle A_0, \mathbf{v_0} \rangle, \langle A_1, \mathbf{v}_1 \rangle, \ldots, \langle A_n, \mathbf{v}_n \rangle\}$, where $A_0 = S$, $\mathbf{v_0} = \mathbf{0}$, and
for all $1 < i < n$:

- $\langle A_i, \mathbf{v}_i \rangle = \langle A_{i-1} + \mathbf{v}_{i-1} + \mathbf{a}, \mathbf{v}_{i-1} + \mathbf{a} \rangle$ where $\mathbf{a} \in \{-1, 0, 1\}^2$
- The line segment from A_{i-1} to A_i does not intersect with any line segment in Con
- A_i must be *within bounds*
- The line segment from A_{i-1} to A_i does not intersect with GL

For $i = n$:

- $\langle A_i, \mathbf{v}_i \rangle = \langle A_{i-1} + \mathbf{v}_{i-1} + \mathbf{a}, \mathbf{v}_{i-1} + \mathbf{a} \rangle$ where $\mathbf{a} \in \{-1, 0, 1\}^2$
- The game is *won* if:
  - the line segment from A_{n-1} to A_n intersects with GL
  - $\overline{A_{n-1}A_n} \cdot \mathbf{CD} > 0$
  - the line segment from A_{n-1} to A_n doesn't intersect with any line segment in Con
  - A_n is within bounds
  - $A_{n-1} \neq S$
- Otherwise, the game is *lost*

If $A_{n-1}A_n \cdot \mathbf{CD} > 0$, the angle between the vectors is below $\frac{\pi}{2}$, meaning that the goal line was crossed in the correct direction. If the starting point selected by the player is on the goal line, the player would win automatically in the first turn. To avoid this, the winning move must start from *another* point than S.

2.4 Formal Problem Definition

In this section we formally define the problem of finding the fastest path through a given VectorRace track. Before doing so we need some definitions.

Definition: Game Strategy

> Given an instance of a VectorRace track, a *game strategy* is a game that is *won*.

A game strategy is an *optimal game strategy* (OGS) if it has the shortest possible length given the VectorRace track. Such an OGS represents the fastest path through the track.

We can now formally define the problem to be solved:

Definition: OGS Problem

> Given an instance of a VectorRace track, find an OGS.

We want to analyze the time (and space) complexity of the OGS problem, and we want to provide an efficient algorithm for solving it.

2.5 Game Strategy Count

Given an instance of a vector race track, how many different game strategies are possible? If only a finite number of strategies existed, and we could list all such strategies, an algorithm for the OGS problem could list all the possible strategies and select the shortest.

Figure 3 on the following page shows a game strategy for a track. The configurations of the strategy are denoted $C_0, C_1, \ldots, C_7$ etc. It is obviously not an OGS. Notice that $C_2 = C_6$ (they are on the same position and they both have the velocity $\mathbf{v} = \left[\begin{smallmatrix} 0 \\ -1 \end{smallmatrix}\right]$).

Another possible strategy is:

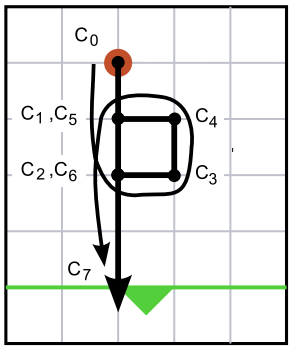


Figure 3: Strategy with duplicate configuration

$$C_0, C_1, \underbrace{C_2, C_3, C_4, C_5}, \underbrace{C_2, C_3, C_4, C_5}, C_6, C_7$$

In fact, an infinite number of different game strategies can be defined by repeating the sequence C_2, C_3, C_4, C_5 any number of times before continuing.

As the number of possible strategies for most tracks is infinite, it is impossible to list them all.

3 OGS Algorithm

3.1 Description

As argued it is not possible to list all possible strategies, and therefore an algorithm for solving the OGS problem needs to take a different approach. Given an instance of a VectorRace track, we can search for an optimal solution using a breadth-first search algorithm. Such an algorithm is presented in this section. The algorithm is named BFS-OGS and is listed as Algorithm 1 on page 7. Please note that remarks are written in *italics*.

The algorithm uses the following procedures:

- **Enqueue**(Q, x) enqueues element x in queue Q
- **Dequeue**(Q) dequeues and returns the first element in the queue Q
- **Add**(S, x) add element x to the sequence S
- **Collision**(T, l) – if $l = \langle A, \mathbf{r} \rangle$ is a line segment and $T = \langle BL, TR, \mathsf{Con}, S, GL, \mathbf{CD} \rangle$ is a track, **Collision**(T, l) returns *true* if:
  - l intersects with any line segment in Con
  - the line segment l crosses GL in the wrong direction (see Section 2.3.2 on page 3)
  - A or $A + \mathbf{r}$ are out of bounds

  and *false* otherwise.
- **Goal**(T, l) returns whether line segment l intersects the goal line on track T

BFS-OGS takes an instance of a vector race track $T = \langle BL, TR, \mathsf{Con}, S, GL, \mathbf{CD} \rangle$ as input and returns an OGS or *null* if no game strategy for the track exists.

The algorithm initially adds the start configuration $\langle S, \mathbf{0} \rangle$ to a set of visited configurations VC, and enqueues a pair consisting of the start configuration and an empty sequence to a queue Q. The sequence P in an enqueued pair $(\langle A, \mathbf{v} \rangle, P)$ contains the configurations on the fastest path from S to A, not including the configuration $\langle A, \mathbf{v} \rangle$ itself as that would be redundant.

The configurations in the queue Q are configurations for which new paths have to be explored. When a pair $(\langle A, \mathbf{v} \rangle, P)$ is dequeued, all possible moves from the configuration $\langle A, \mathbf{v} \rangle$ are examined, and depending on whether the moves are non-colliding, possibly 9 new configurations have to be examined.

These new configurations are enqueued as pairs with the fastest path from S to $\langle A, \mathbf{v} \rangle$ by extending P. When configurations are enqueued in Q, they are also added to VC. The set VC is used to ensure that the same configuration is never enqueued twice.

The algorithm terminates when (i) the goal line is crossed in the correct direction, or (ii) the queue is empty. If the goal line is crossed, the path P to the *winning configuration* is the OGS and is returned. On the other hand, if the queue is emptied no path from S crossing the goal line exists.

3.2 Correctness Proof Outline

Here, I will outline a correctness proof for BFS-OGS. The algorithm should have the following properties – it must:

1. check all legal paths, in order of path length
2. never examine more than one path to any given configuration
3. return a game strategy when it is found
4. return null if no game strategy exists for the given track

Properties 1, 3, and 4 ensures a breadth-first search of every legal path. Property 2 guarantees that the algorithm always terminates, even if no game strategy for the track exists, as there is a finite number of unique configurations on any track. If all 4 properties are satisfied, it follows that the algorithm, given a track, will return the OGS if a game strategy exists, and null otherwise.

The 4 properties are investigated in detail below:

1. In line 3, the starting configuration is enqueued in Q. From that configuration, configurations from all 9 acceleration possibilities are generated in lines 7 and 9. The collision check in line 10 ensures that the illegal ones are discarded. All the legal new configurations are added to Q in line 16. When returning to line 4, all these new configurations just added to Q are checked similarly. When repeating this process until no legal configurations are left, property 1 will be satisfied.
2. Line 2 puts the starting configuration into VC. In line 15, any non-winning configuration is added to VC. Line 10 ensures that only configurations *not* in VC are checked. This satisfies property 2.
3. Lines 11-13 ensures that the path to the goal line (the game strategy) is returned when the goal is reached, and thus property 3 is satisfied.
4. If Q is emptied, the **while** loop drops out to line 21, where null is returned. This is only performed when all legal paths has been investigated due to property 1, and thus property 4 is satisfied.

This concludes the outline of the correctness proof of BFS-OGS.

4 Time Complexity

We analyze the time complexity of the BFS-OGS algorithm in terms of the number of contours on the track, $|\mathsf{Con}|$, and the total number of configurations of the track, TNC. TNC is bounded, as the number of positions on a track is finite and the maximum and minimum velocity at any point also is bounded by the dimensions of the track. The actual value of TNC is investigated in Section 6 on page 9.

First we consider the operations introduced in the algorithm. The **Enqueue**, **Dequeue**

Algorithm 1 BFS-OGS

Require: $T = \langle BL, TR, \mathsf{Con}, S, GL, \mathbf{CD} \rangle$ is a track
Ensure: BFS-OFS returns an OGS or returns null if no game strategy exists for T

```
 1: Q ← ∅
 2: VC ← {⟨S, 0⟩}
 3: Enqueue(Q, (⟨S, 0⟩, ∅))
 4: while Q ≠ ∅ do
 5:    (⟨A, v⟩, P) ← Dequeue(Q)              dequeue a configuration and a path
 6:    P ← Add(P, ⟨A, v⟩)
 7:    for all a ∈ {−1, 0, 1}² do            for all possible accelerations
 8:       l ← ⟨A, v + a⟩                     make line segment
 9:       C ← ⟨A + v + a, v + a              make new configuration⟩
10:       if ¬ Collision(T, l) ∧ C ∉ VC then    if it's OK, and we haven't visited it before
11:          if Goal(T, l) then                 success!
12:             P ← Add(P, C)
13:             return P
14:          else                            just another new configuration
15:             VC ← VC ∪ {⟨A, v⟩}           been here
16:             Enqueue(Q, (C, P))           add it to the queue
17:          end if
18:       end if
19:    end for
20: end while
21: return null      no game strategy exists for this track
```

and **Add** operations can be implemented in constant time if the queue Q and the sequence P used by these operations are implemented as linked lists. **Goal** can be implemented in constant time.

As the **Collision** operation in the worst case has to check for collision with all the contours on the track (the worst case is when the passed line segment does *not* collide with any of the line segments in Con), the operation depends on $|\mathsf{Con}|$.

The final non-trivial operation is the check $C \notin \mathsf{VC}$ in line 10. In the worst case the whole set has to be examined. Thus, the operation is bounded by the maximum size of the set, which corresponds to the maximum number of different configurations the algorithm will examine, i.e. TNC.

We also have to determine how many iterations the **while** loop in line 4 performs in the worst case. The set VC ensures that the same configuration is never enqueued twice, and as exactly one configuration is dequeued in each iteration of the loop, the number of iterations must be TNC in the worst case.

The cost factors are shown in Table 4 on the following page where c denotes constant time. The worst case time complexity of BFS-OGS is

$$\begin{aligned} &O(\mathsf{TNC} \cdot (|\mathsf{Con}| + \mathsf{TNC})) \\ &= O(\mathsf{TNC} \cdot |\mathsf{Con}| + \mathsf{TNC}^2). \end{aligned}$$

Since lookup in the set VC has a worst case time complexity of TNC, a more efficient configuration container data structure could greatly improve performance. As the elements in VC are unique and the total number of configurations is bounded, the set could be replaced by a boolean array; at the start of the algorithm, all values in the array are initialized

c	1: $Q \leftarrow \emptyset$
c	2: VC $\leftarrow \{\langle S, \mathbf{0}\rangle\}$
c	3: Enqueue(Q, $(\langle S, \mathbf{0}\rangle, \emptyset)$)
TNC	4: **while** $Q \neq \emptyset$ **do**
c	5: $(\langle A, \mathbf{v}\rangle, P) \leftarrow$ Dequeue(Q)
c	6: $P \leftarrow$ Add($P, \langle A, v\rangle$)
9	7: **for all** $\mathbf{a} \in \{-1, 0, 1\}^2$ **do**
c	8: $l \leftarrow \langle A, \mathbf{v} + \mathbf{a}\rangle$
c	9: $C \leftarrow \langle A + \mathbf{v} + \mathbf{a}, \mathbf{v} + \mathbf{a}\rangle$
|Con| + TNC	10: **if** $\neg$ Collision(T, l) $\wedge$ $C \notin$ VC **then**
c	11: **if** Goal(T, l) **then**
c	12: $P \leftarrow$ Add(P, C)
c	13: return P
	14: **else**
c	15: VC $\leftarrow$ VC $\cup \{\langle A, \mathbf{v}\rangle\}$
c	16: Enqueue(Q, (C, P))
	17: **end if**
	18: **end if**
	19: **end for**
	20: **end while**
c	21: **return** null

Table 1: BFS-OGS time complexity

to *false*, as no configurations have been visited at this stage. To mark a configuration as visited, the given configuration is converted to an index value for the array, and the array value at the index is set to *true*. A similar procedure is then used to check if a configuration has been visited. If the conversion from configuration to array index is constant time, this array-based configuration container is very time-efficient, as it reduces the overall time complexity of the algorithm to

$$O(\text{TNC} \cdot |\text{Con}| + \text{TNC}).$$

In Section 6 on the next page, we will develop a function that computes a tight upper bound for TNC.

5 Space Complexity

In this section, the space complexity of the BFS-OGS algorithm is analyzed. It is assumed that the track definition is not a part of the space requirements of the algorithm.

In order to analyse the worst case space complexity of BFS-OGS we have to determine the space usage of the data structures in the algorithm, i.e. the queue Q, the set VC, and the paths P.

In the worst case, all possible configurations are added to VC. Thus, the space consumed by VC is bounded by TNC. Each time a configuration is *dequeued* from Q, 9 distinct configurations are in the worst case *enqueued* to Q. This worst case scenario implies that when all configurations are contained in VC, 8 out of 9 configurations are contained in Q. As previously mentioned, the number of configurations is bounded by TNC, thus the number of configurations in Q is $\frac{8}{9}\text{TNC} = O(\text{TNC})$.

Each configuration in Q is paired with a path P. In principle the lengths of these paths can range from 0 to TNC, and as $\frac{8}{9}$TNC paths

in worst case are present in Q this will be very space consuming. However, an important observation is that when 9 new configurations are enqueued, they all share the same path. An efficient implementation could use *references*, such that each enqueued configuration has a reference to its predecessor configuration on the path from the start configuration. This consumes only the space required for one reference for each configuration, and is thus bounded by $O(\text{TNC})$.

Therefore, the worst case space complexity of BFS-OGS using references is bounded by

$$O(\text{TNC}) + O(\text{TNC}) + O(\text{TNC}) = O(\text{TNC}).$$

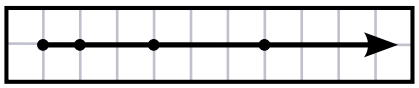
6 Number of Configurations

In this section, we will find an upper bound for TNC for any track, and develop a function TNC() that computes this upper bound from the dimensions of a track[2]. Based on this, we will develop a new configuration container data structure as a replacement for the set VC.

The upper bound for TNC on any track is equal to the number of configurations on an *empty track* with the same dimensions as the given track.

First we observe that horizontal movement is entirely independent of vertical movement; in each turn, a player may accelerate -1, 0 or 1 horizontally as well as vertically. To simplify matters, we investigate an empty one-dimensional track where only horizonal movement is allowed.

[2]Please note that *the value* TNC refers to the number of configurations on the actual non-empty track, whereas *the function* TNC() returns an upper bound for this value



(position 1 is the leftmost possible position)

position	1	2	3	4	5	6	7	8	9	10
v_r	0	1	1	2	2	2	3	3	3	3
NV_r	1	2	2	3	3	3	4	4	4	4

Figure 4: Maximum Right Velocities

6.1 One-dimensional Track

For simplicity, it is assumed that $BL = (0, 0)$ in this section.

The maximum velocity in the right direction (v_r) at position 1 is 0, as the player could only be in the leftmost position of the track if he moved there from the right or is standing still. If the player moves right to position 2, $v_r = 1$. However, the player cannot have a $v_r = 2$ to the left of position 4, because he only can accelerate 1 each turn.

The *number of possible velocities* in the right direction (NV_r) is always $v_r + 1$. For instance, if $v_r = 3$, the possible velocities are $0, 1, 2$ and 3. Figure 4 show this relationship for the 10 first positions. NV_r, the sequence $1, 2, 2, 3, 3, 3, 4, 4, 4, 4,$ etc., is the *self-counting sequence*[3], and is given by:

$$\text{NV}_r(i) = \left\lfloor \frac{1}{2} + \sqrt{2i} \right\rfloor.$$

The number of possible velocities in the left direction are calculated similarly from the maximum velocity in the left direction (v_l). The number of configurations for each position (NC) is the number of right and left velocities added together minus 1 as the velocity 0 is counted twice (see Table 2 on the next page). Thus, the total number of configurations on an empty one-dimensional track with n positions ($\text{TNC1}(n)$) is:

position	1	2	3	4	5	6	7	8	9	10
NV_r	1	2	2	3	3	3	4	4	4	4
NV_l	4	4	4	4	3	3	3	2	2	1
NC	4	5	5	6	5	5	6	5	5	4

Total number of configurations,
$\text{TNC1}(10) = 2\sum_{i=1}^{10} \left\lfloor \frac{1}{2} + \sqrt{2i} \right\rfloor - 10 = 50$
(the sum of the last row in the table).

Table 2: Number of Configurations

$$\text{TNC1}(n) = 2\sum_{i=1}^{n} \left\lfloor \frac{1}{2} + \sqrt{2i} \right\rfloor - n.$$

6.2 Two-Dimensional Track

Using the function TNC1, the number of configurations on an empty two-dimensional track can easily be calculated.

At each position in an empty two-dimensional track, a number of horizontal and vertical velocities are possible. As the horizontal and vertical velocities are independent, the number of configurations at a given position is the number of horizontal velocities multiplied with the number of vertical velocities. It is then easily verified that the number of configurations on an empty $w \times h$ track ($\text{TNC}(w, h)$) is

$$\text{TNC}(w, h) = \text{TNC1}(w) \cdot \text{TNC1}(h).$$

Figure 5 shows calculation of $\text{TNC}(5, 3)$. First, we calculate $\text{TNC1}(3) = 7$ and compare the result with the sum of the last row in the table. Then, we calculate $\text{TNC1}(5) = 17$. The last table shows the number of configurations at each point in the 5×3 track. The sum of these is 119, as is $\text{TNC}(5, 3)$.

Figure 6 shows the relationship between the dimensions of square tracks and TNC(). It is obvious that the space requirements for large tracks are considerable.

Number of configurations on a 3×1 track:

position	1	2	3
NV_r	1	2	2
NV_l	2	2	1
NC	**2**	**3**	**2**

$\text{TNC1}(3) = 2\sum_{i=1}^{3} \left\lfloor \frac{1}{2} + \sqrt{2 \cdot 3} \right\rfloor - 3 = 7$

Number of configurations on a 5×1 track:

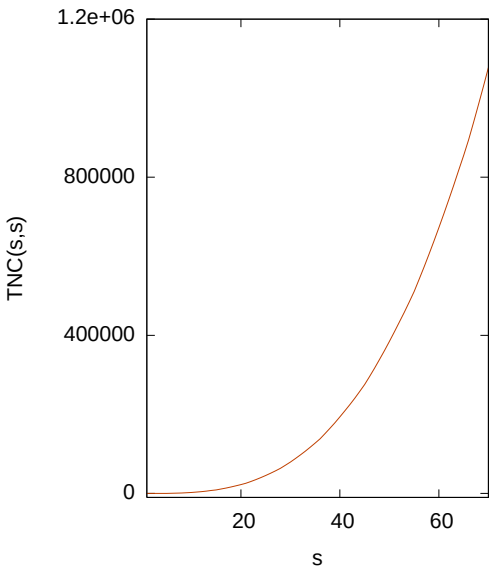
position	1	2	3	4	5
NV_r	1	2	2	3	3
NV_l	3	3	2	2	1
NC	**3**	**4**	**3**	**4**	**3**

$\text{TNC1}(5) = 2\sum_{i=1}^{5} \left\lfloor \frac{1}{2} + \sqrt{2 \cdot 5} \right\rfloor - 5 = 17$

# of configurations		horizontal				
		3	**4**	**3**	**4**	**3**
vertical	**2**	6	8	6	8	6
	3	9	12	9	12	9
	2	6	8	6	8	6

Total number of configurations,
$\text{TNC}(3, 5) = \text{TNC1}(3) \cdot \text{TNC1}(5) = 7 \cdot 17 = 119.$

Figure 5: Configurations on Two-Dimensional Track



This plot shows the relationship between track size and the upper bound for the configuration count on tracks with dimensions $s \times s$.

Figure 6: TNC() as a function of track size

6.3 Configuration Container

This upper bound for the number of configurations can be used to create a configuration container for the BFS algorithm. The container is essentially an two-dimensional array with some extra offset information.

The container depends on storing two lists of offsets, for horizontal and vertical indexing, respectively. Algorithm 2 on the following page calculates the offsets for one of the dimensions. Horizontally, $\mathrm{v}(i) - 1$ represents the maximum velocity in the left direction at position i. Vertically, $\mathrm{v}(i) - 1$ represents the maximum velocity upwards at position i.

The offset lists, here called xOffset and yOffset, are used to index a configuration $\langle A, \mathbf{v} \rangle$ in the configuration container array:

$$\begin{aligned} \texttt{xIndex} &= \texttt{xOffset}[A_x] + v_x \\ \texttt{yIndex} &= \texttt{yOffset}[A_y] + v_y \\ \texttt{visited} &= \texttt{VCArray}[\texttt{xIndex}][\texttt{yIndex}] \end{aligned}$$

This indexing method ensures that there is *exactly one* unique index in the configuration container array for every possible configuration on an empty track.

As a very simple example, consider a one-dimensional track with 5 positions. The following offset list is returned by **createOffsets**:

$$\begin{aligned} \texttt{offset}[1] &= \mathrm{v}(1) = 2 \\ \texttt{offset}[2] &= \texttt{offset}[1] - \mathrm{v}(1) + \mathrm{NC}(1) + \mathrm{v}(2) \\ &= 2 - 2 + 3 + 2 = 5 \\ \texttt{offset}[3] &= \texttt{offset}[2] - \mathrm{v}(2) + \mathrm{NC}(2) + \mathrm{v}(3) \\ &= 4 - 1 + 4 + 1 = 8 \\ \texttt{offset}[4] &= \texttt{offset}[3] - \mathrm{v}(3) + \mathrm{NC}(3) + \mathrm{v}(4) \\ &= 8 - 1 + 3 + 1 = 11 \\ \texttt{offset}[5] &= \texttt{offset}[4] - \mathrm{v}(4) + \mathrm{NC}(4) + \mathrm{v}(5) \\ &= 11 - 1 + 4 + 0 = 14 \end{aligned}$$

Table 3 on the next page shows all possible configurations on an empty one-dimensional track with 5 positions as A_x and v_x. The result of adding the offset list values to the velocity is a sequence of unique array indices.

On a $w \times h$ track, the two lists have a combined space requirement of $O(w + h)$.

The VC array for a track with dimensions $w \times h$ should have the dimensions $\mathrm{TNC1}(w) \times \mathrm{TNC1}(h)$. It follows that the space requirement for container is $O(\mathrm{TNC})$ + the insignificant space consumed by the offset arrays. If the algorithm used a set for VC, the average case space requirement would be less than the worst case, especially if the tracks has contours on it, as this reduces the number of legal configurations on the track. This container always uses the same amount of space, so the average case space requirements are equal to the worst case space complexity, $O(\mathrm{TNC})$. However, the fact that array lookups can be performed in constant time, and that the time complexity of **createOffsets** is insignificant, yields a significant time performance gain.

7 Results

During the writing of this article, an implementation of the BFS-OGS algorithm was created in the Java programming language.

An implementation using the configuration container developed in Section 6.3 had considerably better time performance than an implementation using a Java `ArrayList`-based implementation of a set. Figures 7 and 8 shows this difference in performance.

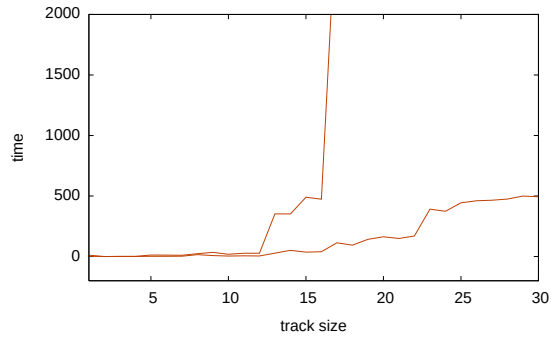
The space complexity of the BFS-OGS algorithm prevents an implementation that can handle very large tracks. The implementation developed during the writing of this article could not handle tracks larger than 200×200 on state-of-the-art computer hardware.

Algorithm 2 createOffsets

```
offset[1] ← v(1)
for i = 2 to size do
    offset[i] ← offset[i − 1] − v(i − 1) + NC(i − 1) + v(i)
end for
return offset
```

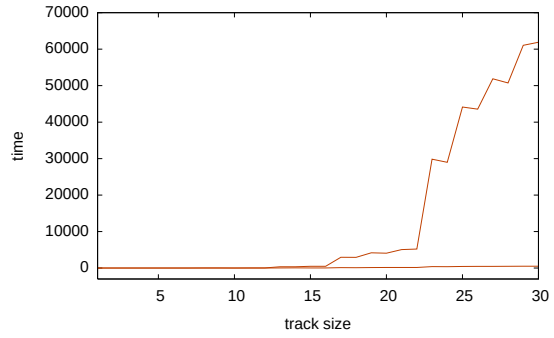
A_x	1	1	1	2	2	2	2	3	3	3	4	4	4	4	5	5	5
v_x	-2	-1	0	-2	-1	0	1	-1	0	1	-1	0	1	2	0	1	2
offset$[A_x]$	2	2	2	5	5	5	5	8	8	8	11	11	11	11	14	14	14
xIndex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Table 3: Computing array indices from configurations



The topmost curve is the performance of the `ArrayList`-based implementation, and the lowermost curve is the performance of the configuration container developed in Section 6.3, both as a function of track size.

Figure 7: Performance of 2 different BFS-OGS implementations



Same results as Figure 7, plotted with a different scale on the y-axis.

Figure 8: Performance of 2 different BFS-OGS implementations - zoomed out

8 Applying Binary Decision Diagrams

The problem of finding the fastest path in a VectorRace track could also be solved using Binary Decision Diagrams (BDDs)[1]. Here, such a BDD solution is outlined.

The BDD approach requires that the problem can be evaluated as a boolean expression, which makes it necessary to convert the OGS problem to a boolean expression. This is accomplished via the following steps:

1. Define the function **Collision** using only whole numbers and basic logic and arithmetic operators. **Collision** must evaluate the formula shown in Section 2.1 on page 2. The formula has fractions, but this is not a problem, as we only need to find out whether t and u are contained in the interval $[0\,;1]$. This problem can be reduced to:

   (a) the whole fraction is negative if and only if *only one* of the numerator and denominator are negative

   (b) the whole fraction is larger than 1 if and only if the numerator is *larger* than the denominator

   If none of the above propositions are *true*,

the fraction is in the interval $[0\,;1]$, and **Collision** can return *false*.

The rest of the problem is trivially converted to whole numbers and basic arithmetic and logic operators.

2. The whole numbers of the arithmetic expressions are converted to binary numbers, which can be expressed as boolean formulas with one variable for each bit in the number. Of course this demands a limit for the number of bits per number and thus a limit of the magnitude of the numbers. The largest number to evaluate can be computed from the dimensions of the track.

3. The arithmetic operators are then expressed as boolean propositions: binary addition, binary multiplication, binary negation, greater-than, and the sign function. As an example, the binary addition proposition (with only 2 bits) would have this form:

$$add(a_0, a_1, b_0, b_1, c_0, c_1)$$

and would return the value of the proposition $a+b=c$ given that the binary digits of a are a_0 and a_1, etc.

The subtraction operator can be created as a combination of addition and negation, and the sign function is needed for the fractions mentioned earlier.

By combining conversions of the collision checks and the remaining rules, a BDD can be algorithmically created for any track, and it can be used to find all paths that evaluate the expression to *true*.

This approach, however, has a problem: the line segment intersection formula described in Section 2.1 on page 2 uses a relatively large number of multiplication operations. A result from Randal E. Bryant's article [2], states that:

> The OBDD representations of the multiplication [...] function are always of exponential OBDD size independent of the chosen order of the input variables.

Thus, a BDD-based solution of the VectorRace problem as defined in this article will most likely *not* be more efficient than the BFS-OGS algorithm, as the BDDs will become too large.

9 Conclusion

A model for the game of VectorRace has been developed and the problem of finding the fastest path from starting point to goal line has been defined. The model is very flexible and almost any conceivable two-dimensional track can be defined.

An algorithm that finds the fastest path through any given track has been created, and it has been analyzed for time and space complexity. Based on this analysis, I have developed a data structure that greatly reduces the time complexity of the algorithm.

I have attempted an application of binary decision diagrams to the problem, but a BDD-based solution is probably less efficient than the BFS-OGS algorithm.

Tests show that the algorithm works and the new data structure greatly improves performance, though the space complexity of the problem prohibits finding fastest paths on tracks with very large dimensions.

10 Future Work

The algorithms and data structures presented in this article can trivially be extended to three dimensions, and thus be used for finding the fastest path through any 3D environment that can be defined discretely.

The BFS-OGS algorithm could be optimized with simple heuristics, such as trying a depth-first search in the direction of the goal line before doing the breadth-first search, thus improving the average-case performance.

The algorithm could also be used to find an approximation of the fastest path through a non-discrete environment.

Acknowledgements

Thanks to Anders Bennett-Therkildsen, Jeppe Carlsen, Andreas Bue Holmgaard Madsen, and Kristian Stougaard Ahlmann-Ohlsen for initial cooperation.

Special thanks to Jeppe and Anders.

Thanks to Hans Schmid for mathematical ideas.

References

[1] Henrik Reif Andersen. An introduction to binary decision diagrams. http://www.itu.dk/people/hra/bdd97.ps, 1998. Lecture notes.

[2] Randal E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. http://www.cs.cmu.edu/~bryant/pubdir/ieeetc91.pdf, 1991.

[3] Eric W. Weisstein. MathWorld, "Self-Counting Sequence", 2005. http://mathworld.com/Self-CountingSequence.html.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001. ISBN: 0-262-03293-7.