Master Thesis

David Kuboň

Genetic Approach To Hypercube Problems

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Petr Gregor, Ph.D.
Study programme: Computer Science
Study branch: Theoretical Computer Science

Prague 2017
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 21 July 2017

David Kuboň
Title: Genetic Approach To Hypercube Problems

Author: David Kuboň

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Petr Gregor, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The main focus of this thesis are hypercubes. In the first part, we introduce hypercubes, which form an interesting class of graphs that has practical uses in networks and distributed computing. Because of their varied applications, the thesis describes the graph-theory problems related to hypercubes such as searching for detour spanners, minimizing their maximal degree and finding multiple edge-disjoint spanners. It also overviews current results on selected hypercube problems and proposes a solution using a genetic algorithm. The genetic algorithm is designed, implemented and its performance is evaluated. The conclusion is that applying a genetic algorithm to some hypercube problems is a viable, but not the most effective method.

Keywords: hypercube genetic algorithms spanning subgraph
I am deeply grateful to my supervisor Mgr. Petr Gregor, Ph.D. for his expertise, time, guidance and patience.

I want to express my gratitude to my colleagues Petr Kubát, who gave me valuable advice on programming in C#, and Natalia Nudga, whose language abilities filled in where I was unsure.

I am thankful to Dorothea Böken for all the hours she spent explaining biology to a complete ignorant. I believe it paid off.

I would also like to thank Melissa Johnston for her styling and grammar advice.
4 Designing a Genetic Algorithm

4.1 Goal .................................................. 40
4.2 Expectations .......................................... 40
4.3 Representation ......................................... 41
4.4 Initialization .......................................... 41
  4.4.1 Random ............................................ 41
  4.4.2 Good Spanner ...................................... 41
  4.4.3 Good Spanner on a Subcube ...................... 41
4.5 Crossover ............................................ 42
  4.5.1 \( n \)-point ........................................ 42
  4.5.2 Subcube Swap ..................................... 42
4.6 Mutations ........................................... 42
  4.6.1 Random ............................................ 42
  4.6.2 Repairing .......................................... 43
  4.6.3 Improving ......................................... 43
  4.6.4 Translation and rotation ......................... 43
4.7 Selectors ............................................ 44
4.8 Fitness .............................................. 45
  4.8.1 3-spanner size .................................... 45
  4.8.2 Maximal degree .................................... 45
4.9 Edge-disjoint subgraphs ............................ 46
  4.9.1 Edge-disjoint 3-spanners ......................... 46
  4.9.2 EDS with longer detours ......................... 46

5 CubeBreeder ............................................ 47

5.1 Data Structures ...................................... 47
  5.1.1 Analysis .......................................... 47
  5.1.2 Representation .................................... 48
5.2 GA Implementation ................................... 50
  5.2.1 Subcube Swap Crossover ......................... 50
  5.2.2 Edge-Repairing Mutation ......................... 50
  5.2.3 3-spanner Property .............................. 51
  5.2.4 Spanning tree ................................... 52
  5.2.5 Parallelism ....................................... 53
5.3 Input / Output ....................................... 54
  5.3.1 Hypercube Structure .............................. 54
  5.3.2 Hypercube Subgraph Description .............. 54

6 User Documentation ................................... 56

6.1 What is configurable ................................. 56
6.2 How to configure ..................................... 57
6.3 Extending the program .............................. 60
  6.3.1 Settings .......................................... 60
  6.3.2 Operators ......................................... 61
  6.3.3 Selector .......................................... 61
  6.3.4 Fitness .......................................... 61
6.4 Interpreting Console Output ....................... 62
6.5 Logging ............................................. 62
Introduction

Hypercubes

Figure 1: Examples of hypercubes $Q_n$ for $n = 1, 2, 3, 4, 5$ generated using [1].

A hypercube is the generalization of a cube into more dimensions. From the geometrical point of view, an $n$-dimensional hypercube $Q_n$ can be defined as a parallelogram of $n$ mutually perpendicular vectors of the same length. From the combinatorial perspective, where we are interested only in nodes and edges, it is a graph where the vertex set is formed by all binary vectors of length $n$ and edges connect the vertices that differ in exactly one coordinate. From another perspective, it can be defined recursively as a Cartesian product of two graphs, where $Q_1$ is the graph $K_2$, i.e. the complete graph on two vertices.

$$Q_n = Q_1 \times Q_{n-1}$$

We see examples lower-dimensional hypercubes in Figure 1 while Figure 2 shows us hypercube graphs for $n \leq 3$.

Figure 2: Illustration of hypercube graphs. Note the labelling of the vertices with fixed-length binary string, which will be used throughout the thesis. Image generated using [1].

The distance between two vertices is the number of coordinates in which they differ. The diameter of a graph $G$ is the maximum distance between any two vertices. Clearly, $d(Q_n) = n$. The crossing number of $G$ is the lowest number of edge crossings of a plane drawing of $G$.

The authors of [2] present a good overview of the basic properties of hypercubes. In [3] and [4] further characterizations are shown.
Applications Use of Hypercubes

Although the definition of the hypercube may sound rather abstract they are used in practice to simulate synchronous distributed systems, as a model of communication in networks and in other applications.

Networks

Hypercube networks are a multidimensional network with two nodes in each dimension. They are used to connect multiple processors with memory modules and accurately route data. The routing works by traversing the network in the $k$-th dimension where $k$ is the least significant non-zero bit. We compute the distance by XORing the nodes’ binary representation and then use the least significant bit containing value 1 as the direction. In this manner, in each step, we get closer to the target node.

MIMD

For a MIMD\(^1\) distributed memory machine with a hypercube system, a processor and a memory module are placed at each vertex. The diameter of the system is the minimum number of steps it takes for one processor to send a message to the processor that is the farthest away. One disadvantage of the hypercube system is that it must be configured in powers of two, therefore a machine must be built so that it could potentially have many more processors than is really needed for the application. For more information we refer to [5].

Further Uses

As we have just seen, hypercubes are valued for their parallelism-related properties. Further examples include parallel computing such as in [6], using hypercube architecture for a distributed database as described in [7] and one that is more related to this thesis is running genetic algorithms distributively on a hypercube — see [8].

The application of hypercubes are not solely limited to computer science. They are also used in biology for genetic code representation [9, 10].

Summary

For more reading on hypercube usage in networks, distributed processing and parallelism, we suggest [11] as introduction and [12, 13, 14, 15] for more details and [16] for a lifesaver.

Problems on Hypercubes

Generally speaking, our interest lies in special properties and subgraphs of hypercubes. The problems that will be focused on throughout the thesis are in relation to the practical uses of hypercubes.

One interesting property to look for is maintaining the same distance or guaranteeing only a reasonable prolonging of paths between vertices of a graph in its

\(^1\)MIMD – multiple instructions, multiple data.
subgraph. The first option is to strictly maintain the same distance, as described in \[17\]. Another approach is to guarantee a distance worse only by a constant – either between any two vertices or only between the adjacent ones. Such subgraphs, called spanners or subgraphs with detours, were studied by \[18, 19, 20, 21\] and will be further explained in Chapter 1.

Considering hypercube architecture, it is useful when none of the nodes becomes overloaded. Reducing the maximal degree of vertices has a positive effect on this issue. This topic was studied in \[18\].

Viewing the same possible issue from the edge-related side, we are interested in subgraphs where a failure of an edge – due to a mechanical error on an actual machine – does not affect the overall performance. In the language of graph theory, this is equivalent to finding several edge-disjoint spanners. More on this topic, although in a more general way, was described in \[22\].

Lastly, even though not a topic discussed in detail in this thesis, it is worth noting that hypercubes have a high crossing number, making practical creation physically more challenging. However, spanners with fewer edges than the graph have also lower crossing number.

Genetic Algorithms

Genetic algorithms (GA), introduced by Holland \[23\], form a robust nature-based meta-algorithm designed to search the space of feasible solutions. They provide genetics-based means to guide the search towards most promising areas. They do not guarantee finding the optimal result, but offer the possibility to reach “fairly good” results in “reasonable time”. They are quite domain specific, but can be used to solve a wide variety of problems, such as quickly finding a sub-optimal solution to NP-complete problems \[24, 25\], machine learning \[26\], scheduling \[27, 28\], combinatorial optimization \[29, 30, 31, 32\], and others.

The general concept is to create an initial population, select individuals for mating, recombine them using crossover, apply mutation, and select the new generation; repeating that until the end criteria are met. Figure 3 shows the outline in a diagram. For a more thorough introduction we suggest \[33\], or newer and a lot more thorough \[34\].

Advantages of GAs

Compared to traditional algorithms, GAs differ in several ways \[35\]. For example, they work with an encoding of a feature rather than the actual feature. To reach results, instead of trying to find the solution by deterministic approaches, they apply probabilistic rules to an initial population of possible solutions, and with each iteration transform them in order to converge to a solution. In addition, GAs are easy to adapt to parallel processing.

GA Parallelism on Hypercubes

There are some clear advantages to hypercube architecture \[8\] when parallelism on evaluating GAs is concerned. Most importantly, conducting most of the operations during a run of a GA can be done in parallel. Therefore they can be
distributed over the whole architecture and thus by reducing computation time significantly increase efficiency.

**Thesis**

We have already explained what makes hypercube problems interesting. We have shown the practical use of these abstract multidimensional objects, especially where parallelism is concerned. We have also noted their interesting connection to biology.

Further, we have presented genetic algorithms, briefly mentioned their advantages and the importance of parallelism for them. Then we got the reader acquainted with GA running on hypercubes.

An intriguing idea directly follows. If hypercube architecture is good for running GAs, could we go the other way around and design a GA that would search for a hypercube architecture?

**Motivation**

We can look at this idea from several angles. First, we have already presented hypercubes and genetic algorithms as two very different concepts that bring promising results when they are cooperating. Second, hypercubes are highly regular graphs where good local properties can often be extended to the whole graph – same as the concept of short schemas of high order exponentially growing in a population (see Section 3.3). We are not the first ones to think so; in [36] genetic algorithms are used for computing some properties for special classes of graphs and in [37–38] the authors describe their approach to solving a specific hypercube problem using GA.
Thesis structure

The thesis begins in Chapter 1 with a brief introduction into graph theory, followed by an overview of the current research in our area of interest. Chapter 2 presents our insight into the topic and a description of the progress we achieved using traditional graph theory methods.

In Chapter 3 we get acquainted with genetic algorithms, while Chapter 4 explores the possibilities to use them for hypercube problems discussed earlier. Chapter 5 presents the program we designed to apply a GA to hypercubes and Chapter 6 follows with the user documentation of the program.

In Chapter 7 we learn about the experiments and the results that were obtained. The Conclusion then summarizes the research that has been explored throughout this thesis.
1. Introduction To Hypercube Detours

In this chapter, we begin in Section 1.1 with an introduction to graph theory, hypercubes and detour subgraphs of hypercubes. In Section 1.2 we give overview of hypercube automorphisms, which we will need later in Chapter 4. Section 1.3 describes 3-spanners, Section 1.4 focuses on spanners with small maximal degree and Section 1.5 presents some results on other kinds of detour spanners. Finally, in Section 1.6 we show some results on edge-disjoint spanners in hypercubes.

Since none of the proofs presented in this chapter are our own work and are available in the cited papers, we included only those that are of constructive nature or are so short they would not be worth the effort to look up elsewhere. The proofs containing constructions of hypercubes with specific properties are important also for the comparison to other theoretical and computational results.

1.1 Preliminaries

What follows is an introduction to several topics that will be discussed further. For a deeper introduction into graph theory we suggest reading [39].

1.1.1 Graph Theory

Definition 1 (Graph). A graph is an ordered pair \( G = (V, E) \) where \( V \) is the vertex set whose elements are the vertices (or nodes) of the graph. \( E \) is the edge set whose elements are the edges, or connections between vertices. If a graph is undirected, individual edges are unordered pairs \{\( u, v \)\} where \( u \) and \( v \) are vertices in \( V \). If the graph is directed, edges are ordered pairs \((u, v)\).

Definition 2 (Subgraph). Given a graph \( G = (V, E) \) and a graph \( G' = (V', E') \). \( G' \) is a subgraph of \( G \) if and only if \( V' \subseteq V \) and \( E' \subseteq E \).

Definition 3 (Spanning subgraph). A spanning subgraph of a graph \( G \) is a subgraph of \( G \) which contains every vertex of \( G \).

Definition 4 (Degree). The degree of a node \( v \), denoted as \( \deg(v) \) is the number of edges incident to it. The minimum degree of a graph \( G \), denoted as \( \delta(G) \), is \( \min_{v \in V(G)} \deg(v) \). Similarly, the maximum degree \( \Delta(G) \) is \( \max_{v \in V(G)} \deg(v) \).

Definition 5 (Path, cycle). A \( u-v \) path is a sequence of edge-connected vertices where each vertex is used at most once. A cycle is a closed path, where \( u = v \).

Definition 6 (Connectedness). If there is a path between any pair of vertices in \( G \), then we say that \( G \) is connected.

Definition 7 (k-regularity). A graph is \( k \)-regular if every vertex has degree \( k \).

Definition 8 (Hamming distance). For a positive integer \( n \) let \([n] = \{1, 2, ..., n\}\). Let \( \oplus \) denote the addition in \( \mathbb{Z}_2^n \) and for \( i \in [n] \) let \( e_i \) denote the vector in \( \mathbb{Z}_2^n \) with number 1 in exactly the \( i \)-th coordinate and 0 elsewhere. Let \( 0 \) and \( 1 \)
denote vectors of all zeros and ones respectively. The Hamming distance of
\( u = (u_1, u_2, \ldots, u_n) \), \( v = (v_1, v_2, \ldots, v_n) \) \( \in \mathbb{Z}_2^n \) is \( d_H(u, v) = |\{ i \in [n] \mid u_i \neq v_i \}| \).

**Definition 9** (Cartesian product). The cartesian product \( G \square H \) of graphs \( G \) and \( H \) is the graph with the vertex set \( V(G \square H) = V(G) \times V(H) \), where two vertices \( (u, v) \) and \( (u', v') \) are adjacent if \( uu' \in E(G) \) and \( v = v' \) or \( u = u' \) and \( vv' \in E(H) \).

**Definition 10** (Neighbourhood). The neighbourhood \( N(v) \) of a vertex \( v \in V \) is the set \( N(v) = \{ u \in V \mid uv \in E \} \).

**Definition 11** (Dominating set). A subset \( D \subseteq V \) is called a dominating set if every vertex \( v \in V - D \) has at least one neighbour \( u \in D \).

**Definition 12** (Diameter). The diameter (radius) of a graph \( G \) is the maximal (minimal, resp.) eccentricity in \( G \). The eccentricity of a vertex is its greatest distance to any other vertex.

### 1.1.2 Hypercubes

There are many ways to define a hypercube. We will name only those relevant to this thesis.

**Definition 13** (Hypercube). The \( n \)-dimensional hypercube \( Q_n \) is an undirected graph with \( V(Q_n) = \mathbb{Z}_2^n = \{0, 1\}^n \) and

\[
E(Q_n) = \{ uv \mid u \oplus v = e_i \text{ for some } i \in [n] \} = \{ uv \mid d_H(u, v) = 1 \}.
\]

If \( u \oplus v = e_i \), the edge \( uv \in E(Q_n) \) is said to have direction \( i \).

**Definition 14** (Alternative definition). \( Q_n = K_2^n = K_2 \square K_2 \square \ldots \square K_2 \) (\( n \)-fold Cartesian product of \( K_2 \)) if \( n \geq 1 \) and \( Q_0 = K_1 \).

**Definition 15** (Subcube). A subgraph \( H \) of a graph \( G \) is a subcube if \( G = Q_n \), \( H = Q_m \), i.e. both are hypercubes, and \( n \geq m \).

![Figure 1.1: Hypercube \( Q_4 \) with a \( Q_3 \) subcube (yellow).](image-url)
Basic properties

- \(|V(Q_n)| = 2^n|.
- \(|E(Q_n)| = n2^{n-1}|.
- \(Q_n\) is \(n\)-regular.
- The neighbourhood of \(u \in V(Q_n)\) is \(N(u) = \{u \oplus e_i \mid i \in [n]\}\).
- \(Q_n\) has \(\binom{n}{k}2^{n-k}\) \(k\)-dimensional subcubes and \(3^n\) of all subcubes.
- \(Q_{n+m} \simeq Q_n \Box Q_m\) for every \(m \geq 1\).

1.1.3 Detours

In this part we introduce the most interesting property of a hypercube within this thesis: the detour subgraph.

**Definition 16.** A spanning subgraph \(H\) of a graph \(G\) is a \((k,t)\)-detour subgraph if \(d_H(x,y) \leq d_G(x,y) + k\) for every \(x, y\) such that \(d_G(x,y) \leq t\), i.e. distances in \(G\) up to \(t\) are increased in \(H\) by (a delay) at most \(k\). Typically, we study \(t = 1\) or \(t = \infty\).

\((k,t)\)-detour subgraphs are also known as:

- \((2,1)\)-detour: a local detour \[20\] subgraph, a 3-spanner \[40\],
- \((2,\infty)\)-detour: a detour subgraph \[19\],
- \((k,\infty)\)-detour: a \(k\)-detour subgraph, a \(k\)-additive spanner \[18\],
- \((2,t)\)-detour: a subgraph with a \(t\)-detour property \[20\].

We are not interested in any detour spanners, but those that have the minimal number of edges. Therefore we define:

\[f_{k,t}(n) = \min\{|E(G)|; G \text{ is a } (k,t)\text{-detour subgraph of } Q_n}\]

Similarly, when studying the maximal degree in a detour subgraph, we want to minimize it:

\[\Delta_{k,t}(n) = \min\{\Delta(G); G \text{ is a } (k,t)\text{-detour subgraph of } Q_n}\]

**Note:** Since \(Q_n\) is bipartite, we restrict to \(k\) even, due to the fact that in a bipartite graph it is impossible to have detours of odd additional length, as with odd length we cannot be in the same partity.
1.2 Automorphisms

Automorphisms are in some sense a symmetry of an object and a way of mapping it to itself while preserving all of its structure and properties.

We will define two types of automorphisms of $Q_n$. It is known that there are no other but their combinations.

**Definition 17** (Isomorphism). An isomorphism of $G$ and $H$ is a bijection $\phi : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $\phi(u)\phi(v) \in E(H)$.

An automorphism is an isomorphism, where $G = H$.

**Definition 18** (Translation, rotation). For $n \geq 1$, $a \in \mathbb{Z}_2^n$, and $\pi$ a permutation of $[n]$

- a translation by $a$ is the map $t_a : u \to u \oplus a$,
- a rotation by $\pi$ is the map $r_\pi : u \to u\pi$, where $u\pi = (u\pi(1), \ldots, u\pi(n))$ for $u = (u_1, \ldots, u_n)$.

Translation can be viewed as flipping the values of some fixed bits, and rotation as a permutation on them.

The following proposition is a well-known fact. It tells us that we need only one rotation and one translation to create any possible automorphism on a hypercube.

**Proposition 1.** For every $g$, an automorphism on $Q_n$ there is a unique $\pi$ permutation of $[n]$ and a unique $a \in \mathbb{Z}_2^n$ such that $g = g_{\pi,a} = r_\pi \cdot t_a$.

1.3 Local Detour Subgraphs of Hypercubes

In this section we overview the known constructions of local detour subgraphs. Let us remind the reader that a local detour subgraph can be also called 3-spanner or $(2,1)$-detour subgraph.

We begin by showing a lemma giving us the size of a dominating set in $Q_n$. It is based on the results in [41], on which we will focus in Section 2.1. For now, the estimated value in Lemma 2 suffices.

**Lemma 2.** [41] $Q_n$ has a dominating set of size $2^n (1 + o(1))$.

Moreover, for $n = 2^r - 1$ where $r$ is an integer, $Q_n$ has a perfect dominating set of size exactly $2^n (1 + \frac{1}{n})$ by a well-known Hamming code. We will use this fact in the following theorem from [20].

**Theorem 3.** [20] For every $n$: $f_{2,1}(n) \leq 3 \cdot 2^n (1 + o(1))$.

**Proof.** Let us represent $Q_n$ as $Q_m \square Q_{n-m}$ for $m = \lceil n/2 \rceil$, where $Q_m$ is induced by the first $m$ and $Q_{n-m}$ by the last $n - m$ coordinates of $Q_n$. Let $D_1, D_2$ be dominating sets of $Q_m$, resp. $Q_{n-m}$ from Lemma [2]. Let $S = S_1 \cup S_2$ where $S_1 = \{(u, v) \in V(Q_n); u \in D_1, v \in V(Q_{n-m})\}$, $S_2 = \{(u, v) \in V(Q_n); u \in V(Q_m), v \in D_2\}$.
Consider an arbitrary edge \((u,v)\) from \(P\) vertices. This comes from the fact that every vertex in \(S\) is adjacent in \(H\) is a dominating set in \(S\). Denote the union of translates of \(S\) dominating sets of \(2\) copies of \(S\) \(\cup \cdots \cup S\) is detoured by \((u,v)\) where \(x \in D_1\) is a vertex dominating \(u\).

What remains is to check the local detour property. An edge \(\{(u,v),(u',v)\} \not\in E(G)\) is detoured by \((u,v),(u,y),(u',y),(u',v)\) where \(y \in D_2\) is a vertex dominating \(u\), and \(\{(u,v),(u',v')\} \not\in E(G)\) is detoured by \((u,v),(x,v),(x,v'),(u,v')\) where \(x \in D_1\) is a vertex dominating \(u\).

**Remark** For \(n = 2^r - 1\) (where \(Q_n\) has a perfect dominating set of vertices) one can find more precise bounds than \(3 \cdot 2^n (1 + o(1))\) edges.

The following theorem from [20] gives us a lower bound on the size of 3-spanners.

**Theorem 4.** [20] \(f_{2,1}(n) \geq 3.2^n(1 - \sqrt{\frac{112}{n}})\)

### 1.4 Local Detour Subgraphs with Small Maximal Degrees

As stated in the Introduction, having a small maximal degree is beneficial for applications. We will present several results on this topic below.

We begin by showing the results from [20] on an upper bound on the maximal degree \(\Delta\) in a \((2,1)\)-detour spanner.

**Theorem 5.** [20] For every \(n\): \(\Delta_{2,1}(n) \leq 1.5 \sqrt{2n} - 1\).

**Proof.** For \(n < 5\) the statement is obvious. Assume \(n \geq 5\) and let \(m = 2^r \in (\sqrt{n/2}, \sqrt{2n})\), \(S = \lceil (n - m + 1)/m \rceil\) for some \(r\). Let us partition \([n]\) into \(m + 1\) parts \(\cup_{i=0}^m P_i\), where \(|P_0| = m - 1, |P_i| \in \{s - 1, s\}\) for \(i = 1, \ldots, m\). Let \(H\) be a subgraph of \(Q_n\) on all edges of direction from \(P_0\). Clearly, it is a disjoint union of \(2^{n-m+1}\) copies of \(Q_{m-1}\). As \(m - 1 = 2^r - 1\), \(Q_{m-1}\) has a partition into perfect dominating sets \(S_1, \ldots, S_m\) that can be found as cosets of a Hamming code. Let \(S_i'\) denote the union of translates of \(S_i\) into all copies of \(Q_{m-1}\) in \(H\). Then every \(S_i'\) is a dominating set in \(H\). Now \(G\) is the union of \(H\) and all the edges of direction from \(P_i\) incident to \(S_i'\) for every \(i \in [m]\).

It remains to be seen that \(G\) is a \((2,1)\)-detour subgraph and count \(\Delta(G)\). Consider an arbitrary edge \((u,v) \in E(Q_n)\setminus E(G)\). The vertices \(u\) and \(v\) differ in
some coordinate $i \in [n] \setminus P_0$, w.l.o.g. $i \in P_1$, which means they belong to different copies of $Q_{m-1}$ that are adjacent in $Q_n$, say $C_u$ and $C_v$. Therefore, the common projection of $u$ and $v$ into $Q_{m-1}$ has a neighbour $z$ in $S_1$. Let $z_u$ and $z_v$ be the copies of $z$ in $C_u$ and $C_v$. Clearly, $z_u$ and $z_v$ are adjacent in $Q_n$. Concluding that $(u, z_u, z_v, v)$ is a path in $G$.

Finally $\Delta(G) = m - 1 + s \leq m - 1 + n/m \leq \sqrt{n/2} + \sqrt{2n} - 1 = 1.5\sqrt{2n} - 1$. \hfill \Box

Next we mention the lower bound from the same paper [20].

**Theorem 6.** [20] For every $n$: $\Delta_{2,1}(n) \geq \sqrt{2n} + \frac{1}{4} - \frac{1}{2}$. 

### 1.5 Other Detour Subgraphs

More general spanners are presented in this next section.

#### 1.5.1 $(2, \infty)$–Detour Subgraphs

The authors of [18] present the following upper bound on 2-additive spanners.

**Theorem 7.** [18] For every $n$: $f_{2,\infty}(n) < 2^n \cdot \frac{3}{4} \sqrt{2n}$.

The same paper shows also a result for the lower bound.

**Theorem 8.** [18] For every $n$: $f_{2,\infty}(n) > 2^n \cdot \frac{\log_2(n)}{2^{10^6}}$.

**Remark** As for the lower bound, the above construction preserves diameter $n$ as proved in [19].

A question about the maximal degree in a 2-additive spanner is answered in [18].

**Theorem 9.** [18] For every integer $k \geq 2$ and $n \geq 21$,

$$\frac{n}{\ln n} e^{-2k} \leq \Delta_{k,\infty}(n) \leq 20 \frac{n}{\ln n} \ln \ln n. \quad (1.1)$$

#### 1.5.2 $(k, t)$–Detours for Higher $k$

For $k$-additive spanners the paper [18] brings additional interesting results formulated in Corollary [13]. However, we begin with an observation from [20].

**Observation 10.** [20] The construction of a $(2,1)$–detour subgraph from the proof of Theorem 3 gives a $(4, \infty)$–detour subgraph.
Therefore this result by [18] follows directly.

**Corollary 11.** [18] For every integer \( k \geq 4 \), \( f_{k,\infty}(n) \leq (3 + o(1))2^n \).

This can be further generalized using the following observation from [18].

**Observation 12.** [18] \( f_{k+2,t}(n+1) \leq f_{k,t}(n) + 2^n \)

The authors of [18] obtained from Observation 12, Corollary 11 and Theorem 3 the following result:

**Corollary 13.** [18]

For every even integer \( k \geq 4 \), \( f_{k,\infty}(n) \leq (1 + 2^{1-k/2} + o(1))2^n \).

For every even integer \( k \geq 2 \), \( f_{k,1}(n) \leq (1 + 2^{2-k/2} + o(1))2^n \).

### 1.5.3 Open Problems

In the section above, we briefly touch on several topics that are not part of this thesis. However, due to the closeness in relations they could be considered for further research. Most of them were formulated more than 20 years ago in [19] and have not been answered yet.

**Conjecture 14.** [19] In the smallest \((2,\infty)\)-detour subgraph of \( Q_n \), at most \( \frac{3}{4} \) vertices have degree less than 3.

**Problem 15.** [19] Find good bounds for \( \min \{|E(G)|; G \text{ is spanner of } Q_n \text{ with diam}(G) = n\} \).

### 1.6 Edge-disjoint Spanners

Among the many interesting properties of hypercubes is the number of edge-disjoint spanners (EDS). The authors of [42] obtained a polynomial algorithm that finds the number of edge-disjoint spanning trees in a connected graph \( G \). For \( Q_n \), an upper bound on this number is \( \lfloor n^2 \rfloor \). For even \( n \), there exists an explicit construction, an example of which can be seen in Figure 1.3. Such construction remains an open problem for \( n \) odd.

In this section, we first show how to construct a set of spanners that contains one good spanner and then move on to the main results on edge-disjoint spanners.

**Definition 19** (d-domatic colouring). A d-domatic colouring of \( G \) is a vertex colouring of \( G \) such that each colour class constitutes a d-dominating set of \( G \).

**Remark** The 1-domatic number of a graph \( G \) is the well-known domatic number of \( G \), denoted as \( dom(G) \).

**Lemma 16.** [43] If \( m \) is a power of 2, then \( dom(Q_m) = m \).

When \( m \) is not a power of two, we get a weaker result.

**Corollary 17.** [22] For any integers \( k \geq 3 \) and \( m \geq 2^\lfloor \log k \rfloor \) the \( m \)-dimensional hypercube \( Q_m \) has a 1-domatic colouring with \( k \) colours.
Figure 1.3: An example of two edge-disjoint spanning trees in $Q_4$. The edges of
the first spanning tree are in green and those of the second one in yellow. Note
that such a construction does not preserve the distance well. The two vertices
marked by blue colour are adjacent in $Q_4$, but have distance 5 in the green
spanner.

We can simply domatically colour a $2^\lceil \log k \rceil$-dimensional hypercube with $k$
colours and then extend this colouring to $Q_m$ repeating it in each copy of the
subhypercube.

A lemma from [44] about decomposition into a cycle follows.

Lemma 18. [44] For even integer $m$ the $m$-dimensional hypercube $Q_m$ can be
decomposed into $m/2$ Hamilton cycles.

Definition 20 (Delay). For graph $G$ and its subgraph $H$, a delay is the difference
between the distance of two vertices in $G$ and in $H$.

Finally, we mention the best-known result from [22] on the number of delayed
edge-disjoint spanners.

Theorem 19. [22] For any integers $k \geq 2$ and $d \geq 4k - 2$, there exists a set of
$k$ edge-disjoint spanners $S_1, S_2, ..., S_k$ of $Q_d$ such that $S_1$ has delay at most $4k - 2$.  

16
2. Constructions Of Local Detour Subgraphs For Small Dimensions

We learned in Chapter 1 what are the best-known asymptotic bounds on the number of edges or maximal degree for local detour subgraphs. However, as our focus in the second part of the thesis will be on genetic algorithms and thus smaller dimensions, the bounds for $n \to \infty$ have little value to us. Therefore we would like to know more detailed values for small dimensions.

Before we begin, let us remind ourselves of the best known asymptotic upper bound on the size of a local detour subgraph achieved in [20] and shown in Section 1.3:

$$f_{2,1}(n) = 3 \cdot 2^n (1 + o(1))$$  \hspace{1cm} (2.1)

For this research we would like to know how big is the error concealed in $o(1)$. In [20] we read that it depends on the size of a dominating set, which is taken from [41]. Therefore we tried to establish a more specific value by analysing the mentioned article in Section 2.1. Later in Section 2.2 we explore the dominating numbers of hypercubes for small dimensions. Section 2.3 shows two good constructions for small dimensions. The chapter concludes in Section 2.4 by comparing known bounds for hypercubes of small dimensions.

2.1 Unit Sphere Packings and Coverings of the Hamming Space

This section covers the relevant parts of [41]. We discovered that the article is only available in Russian, and therefore we decided to adapt the part, which is relevant to our problem, and translate it into English. The full paper in its entirety proves results for alphabets of any (fixed) size and deals also with packings.

**Definition 21 (Covering).** A covering of a set $M$ is a system of sets $D$ such that $\cup_{X \in D} X \supseteq M$.

**Definition 22 (Hamming space).** The space $E^n$ of words of length $n$ over a binary alphabet with metric $d(x,y)$, which is equal to the number of mismatches in words $x$ and $y$ is called a $n \text{ dimensional}$ Hamming space.

Obviously, $E^n$ is actually $\mathbb{Z}_2^n$, but we stick to the original notation.

**Definition 23 (Sphere).** In the space $E^n$ a sphere with radius $t$ and centre in point $z$ is the subset

$$B^n_t(z) = \{x \in E^n : d(x,z) \leq t\}.$$  \hspace{1cm} (2.2)

In the space $E^n$, a sphere with radius $t$ and its centre in point $z$ is a subset of $E^n$: $B^n_t(z) = \{x \in E^n : d(x,z) \leq t\}$. The cardinality of all spheres with radius $t$ is the same and is equal to $|B^n_t(0)| = \sum_{i=0}^{t} \binom{n}{i}$. The density of the family of spheres with radius $t$ with centre from some set $V \subset E^n$ is called

$$\mu(V) = |V||B^n_t(0)|2^{-n}.$$  \hspace{1cm} (2.3)
and is equal to the ratio of the sum of points belonging to the spheres to the total sum of points in \( E^n \). The family of spheres is called a covering if any point from \( E^n \) belongs to at least one of the spheres in the family. From now on we will call covering not the whole family, but the set \( V \) of its centres.

Let us denote \( \mu(n, t) \) the minimal density of covering of the space \( E^n \) with spheres with radius \( t \), omitting \( t \) in the notation whenever \( t = 1 \). Since the density of any covering is not less than 1 we get \( \mu(n, t) \geq 1 \).

Now we will extend the definition of a covering.

**Definition 24.** Family of sets, but not necessarily spheres with the same radius, \( Z_1, Z_2, \ldots, Z_l \) is called covering of space \( E^n \) if and only if any point from \( E^n \) belongs at least one set \( Z_i \). The density of a covering is

\[
\mu(Z_1, Z_2, \ldots, Z_l) = 2^{-n} \sum_{i=1}^{l} |Z_i|.
\]  

Let \( U \) be a set in the space \( E^n \) with the property (homogeneity) that \( u \in U \) implies \( \lambda u \in U \) for any \( \lambda \) from the field \( F_2 \). Therefore a homogeneous set \( U \) is a union of some one-dimensional subspaces – lines traversing through the point 0. On each such line which belongs to \( U \), we choose a non-zero element and denote it \( u_1, u_2, \ldots, u_m \). Then

\[
U = \{0\} \cup \bigcup_{i=1}^{m} \{\lambda u_i : 0 \neq \lambda \in F_2\} \text{ and } m = |U| - 1.
\]  

We say that the set \( V = \{v_1, \ldots, v_l\} \) is a \( U \)-covering if for any \( a \in E_2^n \) the equation

\[
v + u = a, \quad v \in V, u \in U
\]  

has no less than one solution \((v, u)\).

**Theorem 20.** \( [41] \) Let \( V \) be a \( U \)-covering, \( V \subset E^n, U \subset E^n, m = |U| - 1 \). Then for any \( c \in E^n \) the set

\[
X_c = \{x = (x_1, \ldots, x_m) \in E^m : c + \sum_{i=1}^{m} x_i u_i \in V\}
\]  

is a covering of space \( E^m \) with unit spheres.

This theorem brings the following corollary, which can be found in \( [45] \).

**Corollary 21.** \( [45] \) If there exists a covering \( X \) with unit spheres of the space \( E^n \), then there exists a unit-sphere covering \( X' \) of the space \( E^n' \) that has the same density for \( n' = n \cdot 2^j + (2^j - 1) \) where \( j = 1, 2, \ldots \).

The two following lemmas were used before, for example in \( [46] \).

**Lemma 22.** \( [41] \) Let there be a covering \( C' \) of space \( E^n' \). Then for any \( n \geq n' \) there exists a covering \( C \) of the space \( E^n \) with density \( \mu(C) = \mu(C') \frac{1+n}{1+n'}. \)

\textbf{Proof.} Let us choose \( C = \{x \in E^n : (x_1, x_2, \ldots, x_n') \in C'\} \), that is \( C \) is an extension \( C' \) by \( n - n' \) informative symbols. It is clear that \( C \) is also a covering, \( |C| = |C'| 2^{n-n'} \). Therefore \( \mu(C) = \frac{|C|(1+n)}{2^n} = \frac{|C'|(1+n)}{2^m} = \mu(C') \frac{(1+n)}{(1+n')} \). \( \square \)
Now we need to define the following function:

\[ \Theta(n) = (1 + n)2^{-\log(1+n)} \]  

(2.8)

From now on in this section, unless specified otherwise, we will use \( \log \) to denote logarithm with base 2.

If we set the covering \( C' \) to be the Hamming Code \( H_i \) of length \( n' = 2^i - 1 \), where \( i = \log(1+n) \), in Lemma 22 we get the inequality \( \mu(n) \leq \Theta(n) < 2 \).

**Lemma 23.** \([41]\) Let there be covering \( C'' \) of space \( E^n \), \( n \geq n'' \) and \( \Theta(n) \geq \Theta(n'') \). Then there exists covering \( C \) of space \( E^n \), for which \( \mu(C) = \mu(C'') \Theta(n) \Theta(n'') \).

**Proof.** Let \( 2^{a-1} - 1 \leq n < 2^a - 1 \) and \( 2^{b-1} - 1 \leq n'' < 2^b - 1 \). As \( n \geq n'' \), we have \( a \geq b \). According to Corollary 21 for \( n' = n'' \cdot 2^{a-b} + 2^{a-b} - 1 \) there exists covering \( C' \) of space \( E^a \) with density \( \mu(C') \), and \( \Theta(n') = \Theta(n'') \), because \( 2^{a-1} - 1 < n' \leq 2^a - 1 \) and \( \Theta(n') = \Theta(n'') \). As function \( \Theta \) is increasing in the interval \( \left[ 2^{a-1} - 1, 2^a - 1 \right) \), we get \( n \leq n' \) and according to Lemma 22 there exists covering \( C \) of \( E^n \) with density no greater than \( \mu(C'') \Theta(n) \Theta(n'') \).

For the remaining part of this section let us fix \( R \) to be a power of a prime number, \( l \) a natural number and \( 2^l \neq R \), \( s = s(l, 2, R) = \lfloor l - \log R \rfloor \), \( n_i = n_i(2, l, R) = (2^i - 1)(R^i - 1)/(R - 1) \) and \( L_i = L_i(2, l, R) = n_i + 1, i = 0, 1, 2, \ldots \).

**Lemma 24.** \([41]\) For an arbitrary natural number \( n \geq n_s \) it holds the following bound on the size of the covering:

\[ \mu(n) < (2^l - 1)2^l/(R - 1)R \text{ for } R < 2^l \]  

(2.9)

**Proof Idea.** For \( s = 0 \) and \( s = 1 \) it is an alternative formulation of Lemma 22 and we require nothing more. \( \square \)

We know from \([47]\) that for an arbitrary constant \( \beta \in [11/20, 1] \) there exists number \( N(\beta) \) such that if \( x \geq N(\beta) \), then there is a prime between \( x \) and \( x + x^\beta \). From this fact and from Lemma 24 we get \( \lim_{n \to \infty} \mu(n, 2) = 1 \) (It is enough to set for \( R \) a prime from interval \( (2^l - 2^\beta, 2^l) \) for \( l \to \infty \)). Now we will set an estimate to the speed of convergence of the density of the best covering to 1.

**Theorem 25.** \([41]\) If \( 11/30 \leq \beta < 1 \), \( \ln \ln n \geq (1 - \beta) \ln(4 \cdot N(\beta)) \) and \( \ln n \geq 2 \cdot 2^{1-\beta}(2 + \frac{(\ln \ln n)(\ln 2)}{(1-\beta)})^2 \) then

\[ \mu(n) < 1 + \frac{2 \cdot 2^{1-\beta} \cdot \frac{(\ln \ln n)(\ln 2)}{(1-\beta)} + 2}{\ln n}. \]  

(2.10)

**Proof.** For a proof we refer to \([41]\). \( \square \)

In Theorem 25 we learned the actual value of the error for Theorem 3. That gives us a much better estimate on the upper bound for higher dimensions.
2.2 Dominating sets

The results from Section 2.1 do not really help with small dimensions, as their estimate of the dominating set is asymptotic. However, the exact values are known for small dimensions, which will be presented in this section.

**Definition 25** (Domination number). The domination number $\gamma(G)$ of a graph $G$ equals the minimum cardinality of a dominating set in $G$.

Not that much is known about the domination numbers of hypercubes. For $n \leq 9$ exact values are known, as well as for $n = 2^k - 1$ and $n = 2^k$. Clearly, for $k \geq 1$ and $2^k - 1 \leq n < 2^{k+1} - 1$ we have $\gamma(Q_n) \leq 2^{n-k}$.

From [48, 49], we obtained Table 1 which shows the known values of domination numbers of small dimension hypercubes. It is worth noting this is a very difficult problem and so far exact values are only known for $n \leq 9$ [49].

<table>
<thead>
<tr>
<th>$n$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
<th>$8$</th>
<th>$9$</th>
<th>$10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(Q_n)$</td>
<td>$1$</td>
<td>$2$</td>
<td>$2$</td>
<td>$4$</td>
<td>$8$</td>
<td>$12$</td>
<td>$16$</td>
<td>$32$</td>
<td>$62$</td>
<td>$107 - 120$</td>
</tr>
</tbody>
</table>

**Table 1**: Sizes of dominating sets in hypercubes of small dimension.

We now need to establish bounds for higher dimensions. The following theorem gives us a very rough estimate for these bounds.

**Theorem 26.** [49] If $G$ is bipartite graph then $\gamma(G \Box K_2) \leq 2\gamma(G)$.

For some values, even in higher dimensions, we can compute more accurate results as per the following theorem.

**Theorem 27.** [49] If $k \geq 1$, then $\gamma(Q_{2^k-1}) = 2^{2^k-k-1}$ and $\gamma(Q_{2^k}) = 2^{2^k-k}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$15$</th>
<th>$16$</th>
<th>$31$</th>
<th>$32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(Q_n)$</td>
<td>$2048$</td>
<td>$4096$</td>
<td>$67108864$</td>
<td>$134217728$</td>
</tr>
</tbody>
</table>

**Table 2**: Values of $\gamma(Q_n)$ for example dimensions $n = 2^k - 1$ and $n = 2^k$.

In Section 1.3 we learned an upper bound on the number of edges in a local detour spanner is for every $n$

$$f_{2,1}(n) \leq 3 \cdot 2^n (1 + o(1))$$  \hspace{1cm} (2.11)

We know from the proof that this result uses the value of

$$\frac{2^n}{n} (1 + o(1))$$  \hspace{1cm} (2.12)

as an estimate of the size of a dominating set. Combining the inequality (2.11) and (2.12), we get the following formula for an upper bound:

$$f_{2,1}(n) \leq 3 \cdot 2^n (1 + o(1)) = \frac{3 \cdot n}{n} 2^n (1 + o(1)) = 3 \cdot \gamma(Q_n)$$

Using values from Table 1 we calculate upper bounds as in Table 3.

These results are considerably better than the asymptotic ones. With future development of our knowledge of dominating sets, we could improve these results further. More detailed information on dominating sets can be found in [50].
Table 3: Values for the upper bound on the number of edges in the minimal local detour spanners in hypercubes of small dimension.

<table>
<thead>
<tr>
<th>n</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{2,1}$</td>
<td>18</td>
<td>48</td>
<td>120</td>
<td>216</td>
<td>336</td>
<td>768</td>
<td>1674</td>
<td>3600</td>
</tr>
</tbody>
</table>

2.3 Two 3-spanner constructions for small $n$

This section of the thesis presents two constructions from [51]. These two constructions give us efficient 3-spanners for small dimensions. However, these 3-spanners use more than a linear number of edges with respect to the number of vertices.

Lemma 28. [51] For $n \geq 3$, the hypercube $Q_n$ has a 3-spanner with $2^n(n+5)/8$ edges.

Proof. Let $n = 3$ and let $A \subseteq E(Q_3)$ be the set of all edges incident with the Hamming code $H_3 = \{000, 111\}$. We see that the set

$$E(S) = A \cup \{\{010, 110\}, \{001, 101\}\}$$

is a 3-spanner $S$ of $Q_3$ with 8 edges.

Let $n > 3$, then the hypercube $Q_n$ can be viewed as a Cartesian product of $Q_3$ and $Q_{n-3}$. In each copy of $Q_3$ we take a 3-spanner $S$ as described above and add two full copies of $Q_{n-3}$ corresponding to vertices in $H_3$. We get a 3-spanner of $Q_n$ with

$$8.2^{n-3} + 2(n-3)2^{n-4} = 2^n \frac{n+5}{8}$$

edges.

It is not a coincidence that we used $Q_3$. Hypercubes of size $n = 2^k - 1$ have very suitable properties for such a construction as they have perfect dominating sets. Therefore we proceed with a construction based on $Q_7$. For this the Steiner triple system will be used.

Definition 26 (Steiner triple system). Let $X$ be a set of $v \geq 3$ elements together with a set $B$ of 3-subset (triples) of $X$ such that every 2-subset of $X$ occurs in exactly one triple of $B$. Then $B$ is called a Steiner triple system.

Lemma 29. [51] For $n \geq 7$, the hypercube $Q_n$ has a 3-spanner with $2^n(n+21)/16$ edges.

Proof. For $n = 7$, we have the Hamming code and the Steiner triple system:

$$H_7 = \{1110000, 0101010, 0011001, 0100110, 0000010, 1010101, 1001010, 0001100, 0110110, 1100101, 0111110, 1011101, 1111110\},$$

$$L_7 = \{\{1, 2, 3\}, \{2, 4, 6\}, \{3, 5, 6\}, \{3, 4, 7\}, \{1, 4, 5\}, \{1, 6, 7\}, \{2, 5, 7\}\}.$$

Since we need a value for an upper bound, we must use the upper bound of the estimate on the domination number.
Let us define permutations $s$ and $z$ of $[7]$ according to the following table

\[
\begin{align*}
i &= 1, 2, 3, 4, 5, 6, 7, \\
s(i) &= 2, 4, 6, 3, 1, 7, 5, \\
z(i) &= 3, 6, 5, 7, 4, 1, 2.
\end{align*}
\]

Note that $L_n = \{\{i, s(i), z(i)\} \mid i \in [7]\}$. Let $A$ be the set of all edges incident with the Hamming code $H_n$. For $i \in [7]$ define sets

\[
B_i = \{\{u \oplus e_{s(i)}, u \oplus e_{s(i)} \oplus e_i\} \in E(Q_7) \mid u \in H_7\},
\]
i.e. $B_i$ is the set of edges with direction $i$ from $H_n \oplus e_{s(i)}$. Observe that $s(i)$ could be replaced with $z(i)$ in the definition of $B_i$ since $H_n \oplus e_i \oplus e_{s(i)} \oplus e_{z(i)} = H_n$ for all $i \in [7]$. Put $B = \bigcup_{i=1}^{n} B_i$. We claim that the set $E(S') = A \cup B$ forms a 3-spanner $S'$ of $Q_7$ with 224 edges.

Let us assume there is an edge $\{x, y\} \in E(Q_7) \setminus A$. Therefore $x, y \notin H_7$. Let $u$ and $v$ be the neighbours of $x$ and $y$ from $H_7$. As $d_{Q_7}(u, v) = 3$, there exists exactly one $i \in [7]$ such that $u \oplus v = e_i \oplus e_{s(i)} \oplus e_{z(i)}$. We denote $C$ the subcube of dimension 3 in $Q_7$ between $u$ and $v$. Let us consider the subgraph $S' \cap C$. It contains edges

\[
E(A \cap C) = \{\{w, w \oplus e_j\} \mid w \in \{u, v\}, \ j \in \{i, s(i), z(i)\}\}, \ \text{and}
\]
\[
E(B_i \cap C) = \{\{u \oplus e_{s(i)}, u \oplus e_{s(i)} \oplus e_i\}, \{v \oplus e_{s(i)}, v \oplus e_{s(i)} \oplus e_i\}\}.
\]

We can see that $S' \cap C$ is isomorphic to the 3-spanner $S$ of $Q_3$ from Lemma 28. Concluding that $d_{S'}(x, y) = d_S(x, y) \leq 3$ since $x, y \in V(C)$. Counting the number of edges we have $|E(S')| = 14|H_7| = 224$.

For $n > 7$, the hypercube $Q_n$ can be considered as a Cartesian product of $Q_7$ and $Q_{n-7}$. In each copy of $Q_7$ let us take the 3-spanner $S'$ and add 16 full copies of $Q_{n-7}$ corresponding to vertices in $H_7$. We obtain a 3-spanner of $Q_n$ with

\[
224.2^{n-7} + 16(n-7)2^{n-8} = 2^n \frac{n + 21}{16}
\]
edges. \[\square\]

Comparing Lemma 28 and Lemma 29 observe that we get 3-spanners of $Q_n$ with the fewest edges using Lemma 28 when $3 \leq n \leq 11$ and using Lemma 29 when $11 \leq n$. Note that when $n = 11$, both Lemma 28 and Lemma 29 perform equally well.

A further improvement of the general upper bound seems difficult with the methods used so far. However unlikely, there is a small possibility that a similar construction could arise for $n = 15$, which is the next dimension with a perfect dominating set.
Figure 2.1: A 3-spanner of $Q_7$. The edges between the copies of $Q_5$ are all in the spanner, but are omitted in the picture, except the edges from vertices $u$ and $v$. The grey areas represent all edges between two consecutive levels of $Q_5$. Note that this is an example of a 3-spanner that is not an additive 4-spanner. The picture is courtesy of [51].

2.4 Comparison

To conclude the graph-theoretical part of this thesis concerning local detour subgraphs, we briefly compare the different results on the upper bound for small dimensions.

First, by comparing the values obtained in Theorem 1 and Lemma 28 we get Table 4. With no surprise, it is evident that the construction designed specifically for a small dimension is considerably better than the asymptotic one.

<table>
<thead>
<tr>
<th>$n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamburger et al. [20]</td>
<td>18</td>
<td>48</td>
<td>120</td>
<td>216</td>
<td>336</td>
<td>768</td>
<td>1674</td>
<td>3600</td>
</tr>
<tr>
<td>Gregor [51]</td>
<td>8</td>
<td>18</td>
<td>40</td>
<td>88</td>
<td>192</td>
<td>416</td>
<td>896</td>
<td>1920</td>
</tr>
</tbody>
</table>

Table 4: Comparison of values for upper bound on local detour spanners in hypercubes of small dimension

Moving to higher dimensions, between $n = 16$ and $n = 31$ the asymptotic theorem takes over as seen in Table 5. However, the exact dimension is not known, because we do not know exact values for $\gamma(Q_n)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>15</th>
<th>16</th>
<th>31</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamburger et al. [20]</td>
<td>92 160</td>
<td>196 608</td>
<td>6 241 124 352</td>
<td>12 884 901 890</td>
</tr>
<tr>
<td>Gregor [51]</td>
<td>73 728</td>
<td>151 552</td>
<td>6 979 321 856</td>
<td>14 227 079 168</td>
</tr>
</tbody>
</table>

Table 5: Comparison of values for upper bound on local detour spanners in hypercubes of some higher dimensions

For higher dimensions the asymptotic construction is strictly better so further comparison makes little sense.
3. Genetic Algorithms

This chapter introduces genetic algorithms. Section 3.1 gives a general introduction and Section 3.2 presents the biological approach to genetics. Section 3.3 follows with theoretical results, while Section 3.4 provides a more detailed explanation of a simple genetic algorithm. This chapter concludes in Section 3.6 by showing the described concepts using a practical example.

3.1 Introduction

Evolutionary algorithms (EA) are one of artificial intelligence’s way of using biological principles and concepts to solve difficult computational problems. It is a generic stochastic population-based meta-heuristic search-based technique. It applies methods from biological evolution to improve individuals, which represent candidate solutions. Environmental and parental selection guides the search in the right direction, while operators – crossovers and mutations – take care of variation in the population of candidate solutions. EAs are a robust meta-algorithm, meaning there is no general algorithm to solve all problems, but the algorithms need to be problem-specific, adapting both the representation and the operators to a specific domain.

A genetic algorithm is the most popular type of EA. It seeks a solution in the form of a string by applying the evolutionary operators. It is often used in optimization problems, in research as well as in machine learning.

Some of the main advantages of using an EA include their speed and efficiency, good parallel capabilities and the ability to deal with a big variety of problems. Evolutionary algorithms do not provide a single best solution to a problem, but multiple solutions. EAs can be terminated during the computation and will always provide a solution. They are very useful when the search space is large and there are many parameters involved. Unfortunately, EAs are not suitable for all problems that may arise.

For instance repeatedly calculating a fitness value, which quantifies the quality of a particular individual, may be computationally expensive in some situations. Moreover, there is no guarantee on the optimality of the solution, especially when this solution is not implemented properly.

Genetic algorithms (GA) are able to provide usable “good enough” solutions at a reasonable speed when calculating an actual optimal result would be very difficult – for instance NP-hard problems. GAs also work on complicated fitness landscapes (see Figure 3.2) where gradient-based methods fail, due to the fact that gradient methods suffer from getting stuck in local optima.

3.2 Biological Perspective

Deoxyribonucleic acid (DNA) is a macromolecule consisting of two anti-parallel poly-nucleotide strands, composed of the four nucleobases adenine, thymine, guanine and cytosine, which form complementary pairs. Triplets of nucleotides, codons, encode the 20 standard amino acids according to the rules set by the
genetic code. This genetic code is degenerate, meaning that one amino acid can be specified by multiple codons, but not ambiguous, meaning that one codon can only specify a single amino acid. Amino acids can bond together forming long, linear chains. Proteins consist of at least one of those long amino acid chains and form the enzymes and macromolecules that are active in biochemical processes and cellular structures. Each protein has its own amino acid sequence that is encoded by the nucleotide sequence of the respective gene. Some genes have multiple variants located in the same position – the alleles. The genetic make-up of an organism, either of all its genes or of a single locus only, is referred to as the genotype. The observable trait, the phenotype, however, is not only determined by the genetic information but also influenced by environmental factors. It is worth noting that a gene does not directly map to a trait – many traits are encoded by multiple genes.

In the cell, DNA is densely packed and organised into long structures, the chromosomes, that contain all or most of the organism’s genetic information; organisms have a fixed number of chromosomes. Genes coding for the same trait are always located at the same position (locus) in the chromosome. Genetic recombination is an important phenomenon taking place during sexual reproduction, providing the offspring with new combinations of traits that cannot be found in either parent. This involves the pairing of homologous chromosomes, allowing the exchange of genetic material between those, also referred to as chromosomal crossover. Here the genes having the same locus in the chromosome is very important since during crossover it guarantees that only corresponding parts of the sequences will be exchanged. The genome is the set of all genetic information in an organism. It is worth noting that a gene does not directly map to a trait – many traits are encoded by multiple genes.

Basis for biological inheritance is the replication of DNA, during which the double-stranded DNA is separated into two strands. Each strand serves as a template for the synthesis of the counterpart, also referred to as semi-conservative replication. Synthesis of the new strand is carried out by the DNA polymerases. Most DNA polymerases themselves are highly accurate. To even further increase the fidelity of DNA replication, several other mechanisms ensure the correct replication of the DNA, e.g. a proofreading ability of some of the DNA polymerases and post-replication mismatch repair mechanisms that can recognise the older strand and repair occurring base mismatches in the newly synthesised strand.

Errors during the replication often result in point mutations. Those can either be indels (insertion/deletion) or substitution of a single nucleotide base in the sequence. The impact of such a mutation can vary from a silent mutation to catastrophic. Since the insertion or deletion of a nucleotide base leads to scrambling the amino acid triplets, such a mutation leads to a so called frameshift and is always catastrophic as the reading frame gets shifted.

The impact of a substitution depends on several factors. Due to the degeneracy of the genetic code a substitution does not necessarily change the amino acid encoded by the mutated codon, a silent mutation. A so called non-sense mutation is the gain or loss of start- or stop-codons, that tell the protein synthesising enzymes where to start/to terminate the protein synthesis.

The other type of such a mutation is a missense mutation where the encoded amino acid is different to the original. In the case that this mutated amino acid...
has very similar properties to the original amino acid and is not located at a position important for the protein’s folding and function, the amino acid change is conservative and the synthesised protein is still functional or might even show an improved function. In a non-conservative change the changed amino acid differs a lot from the originally encoded amino acid, usually having an effect on protein function. It might either lead to a gain of function or activation of the protein, or a loss of function, both often resulting in disease in the organism.

A Mapping to GA

In genetic algorithms, we usually use the term chromosome when referring to a solution. Each solution, depending on its representation, comprises of some building blocks — genes. Alleles refer to the multiple values each gene can have — 0 or 1 in a bit string representation of the chromosome, any number between 0 and $n$ in an integer representation, or others. By genotype we typically refer to the configuration of the chromosome and the term phenotype is not very common in GA.

3.3 Schema Theorem

Not much of Genetic Algorithms is supported by theoretical results. The only exception is the Theory of Schemas, which tries to provide theoretical background to binary representation and operations.

**Definition 27 (Individual).** An individual is a word in the alphabet $0,1$.

**Definition 28 (Schema).** A schema is a word in the alphabet $0,1,\ast$, where $\ast$ denotes either 0 or 1.

A schema represents a set of individuals. A schema with $\ast$ at each of its $r$ positions represents $2^r$ individuals, while an individual of length $m$ can be represented by $2^m$ schemas. There are in general $3^m$ schemas of length $m$. In a population of size $n$ there are $2^m$ to $n2^m$ schemas.

**Definition 29 (Order of schema).** The order of schema $S$, denoted as $o(S)$, is the number of zeros and ones (the number of fixed positions).

**Definition 30 (Defining length).** The defining length of schema $s$, denoted as $d(S)$, is the distance between the first and last fixed position.

**Definition 31 (Fitness of schema).** The fitness of schema, $F(S)$, is the average fitness of all individuals represented by $S$ in a population.

**Note.** The fitness of a schema is population-dependent.

**Theorem 30 (Schema Theorem).** Short schemas with fitness above average and low order experience exponential growth in a population during a GA.

For proof see [33].

**Hypothesis 31 (Building Block Hypothesis).** A genetic algorithm seeks optimal performance through comparison of short, low-order, high-performance schemas, called the building blocks.
There are, however, at least three counter-arguments to the building block hypothesis:

- **Collateral Convergence**: Once the population begins to converge, even slightly, it is no longer possible to estimate the static average fitness of schemas using the information present in the current population.

- **Fitness variance within schemas**: In populations of realistic size, the observed fitness of a schema may be arbitrarily far from the static average fitness, even in the initial population.

- **Compositionality**: Superposition of fit schemas does not guarantee larger schemas that are more fit and these are less likely to survive.

Consequences

Before exploring the consequences of the Theory of Schemas, it must be noted that the theorem is described in terms of expectations for infinite populations. Due to the fact that we work predominantly with finite populations, we must take into account the possibility of sampling errors. That is, if there exists a trait strongly outperforming all others, but no individual has it at the beginning and it cannot be created by a crossover or mutation, the best possible result will never be reached by using genetic algorithms.

- **Coding matters.** Encoding the problem into the individuals in a wrong way will have direct consequences on the efficiency of the algorithm.

- **Size matters.** Simply said, if the search space is too big, it can hardly be explored.

- **Early convergence.** By allowing the GA to converge quickly to the nearest optimum, it is likely that it will get stuck and not search for a different result. This is due to individuals matching this optimum quickly taking control of the population.

The conclusion is that there are good and bad problems for EA and GA.

### 3.4 Simple Genetic Algorithm

For a given problem and a **representation** of solutions, a simple genetic algorithm (SGA) outlines the computation as found in Figure 3.4.

An initial population is created. The next step is to calculate the **fitness** of each chromosome. This process is repeated on a continuous loop until a provided **termination** criterion is met. Within the loop, which represents one generation, **selection** is used to choose parents, **crossover** to cross the parents and **mutation** to mutate the new population members. Environmental selection then decides which members of the population will survive and replace the old population.

Now, an overview on methods and techniques for each of the SGA events is presented. There are many different options and the list is by no means complete.
Create an initial population;

\[
\text{for } i \leftarrow 0 \text{ to } \text{maxGeneration} \text{ do}
\]
  Select parents;
  Crossover;
  Mutation;
  Environmental selection;
end

\[\text{Figure 3.1: A simple genetic algorithm idea in a pseudo-code.}\]

### 3.4.1 Fitness

Fitness function evaluates how “fit” the individuals are to be a solution to the task at hand. It is computed very often during the run of a GA and therefore needs to be sufficiently fast. Sometimes it cannot be computed directly – for instance if the fitness depends on a result of the individual in a competition within the population.

The fitness function forms a landscape, the fitness values of solutions are points in this landscape and the operators move them around. An example of such a landscape for a single objective function is in Figure 3.15 and for a multi-objective 2-dimensional function in Figure 3.2.

\[\text{Figure 3.2: The fitness landscape of a multi-objective fitness function.}\]

A similar concept to the fitness function is the objective function, which is typically used whenever the computed fitness does not have meaningful values for understanding the computation.

### 3.4.2 Representation of Individuals

As with most things in genetic algorithms, choosing the correct representation fitting a particular problem is essential to the efficiency of the algorithm. For example, it may seem that swapping integers and their binary representations does not create any difference in the solution; however the outcome depends on how the operators are chosen. Operators have the ability to create a strong effect on the solution, and need to be chosen carefully.

Section 3.4.2 takes a look at common representations, followed by a brief discussion of their advantages and disadvantages. An example of a possible efficient usage of the representations follows in Section 3.6.
Binary The first way to represent individuals, introduced by Holland, was a vector of binary values — see Figure 3.3. The advantage of using binary values is that we have theoretical results (see Section 3.3) On the other hand, such representation can often be unnatural and inefficient. Other representations, such as using an integer, can be naturally encoded into a binary one. In this instance, the operators for mutation and crossover used in ordinary binary representation do not work in the same way – for instance, flipping a bit might have different effect if it is the first sign bit of an integer or any other. Therefore all bits in such representations should have the same importance or the operators should reflect any imbalances.

![Figure 3.3: Example of binary representation](image)

Integer A vector of integers is another method that is often used. One of the advantages of using integers is that it allows easy representation of a permutation. See Figures 3.4 and 3.5 for examples on these two representations. During the situations when there are multiple (non-binary) alleles for each gene, a multi-character alphabet can also be used.

![Figure 3.4: Example of integer representation](image)

![Figure 3.5: Example of permutation representation](image)

Floating point When we need continuous and not discrete variables, using real values in the representation (Figure 3.6) is the most natural. We are, however, limited by the precision of such representation in the computer. Also, some operators do not work well – for instance, flipping a bit in different parts of a floating point number has significantly different effect.

![Figure 3.6: Example of floating point representation](image)

Others There are many other ways of representing an individual. We will not go into detail, as most of them are highly sophisticated and way beyond the information useful for this thesis. To name a few, trees, matrices, neural networks, finite automata, etc., are used.
3.4.3 Initialization

The method used to initialize the individuals in the starting population will considerably affect the behaviour of the GA. First of all, with some combinations of operators there may be a building block that can be never created, unless already present in the population at least once. Second, if there is not enough variety in the population, the GA will likely get stuck around the most represented area of the search space. Last, if the population is too random, it will take longer for the population to converge to an optimal value.

Unfortunately, there is no general way to approach this problem. The initialization method, like anything else in the GA, has to be chosen carefully depending on the domain and specifics of the solved task. There are two main concepts:

- **Random** – We initialize the population randomly.
- **Heuristic** – We use knowledge of the problem to get a head-start. Knowledge may be limited in certain problems. However, we should be careful not to initialize the whole population heuristically. This would cause a loss in population diversity.

3.4.4 Selection

Selection is the process of choosing individuals for breeding (applying the crossover operator) and for survival in the environment. Generally speaking, the fitness function is evaluated for each individual. Then from the current population, new parents are chosen stochastically, but usually reflecting their fitness. If it makes sense for the specific selection method, the fitness values are then normalized by dividing the fitness value of each individual by the sum of all fitness values.

Another approach is **age-based** selection, when each individual has a set lifespan after which it dies no matter its fitness.

There are other selection algorithms that limit their selection of individuals to those with a higher fitness value than a given (arbitrary) constant.

Selection occurs twice during each generation of a GA. First, during **parental selection** before applying operators, the future parents are selected. Second, in **environmental selection**, which happens after the application of operators. The members of the new population are chosen from the offspring and possibly also from the old population.

In the following paragraphs some of the selection methods and concepts are further explored to offer a greater overview and understanding.

**Methods**

By methods we mean actual selection mechanisms that can be used on their own.

**Roulette Wheel** We could imagine this selection method as a roulette wheel in a casino. There, the chance to get every number is the same; here, the chances of an individual to be selected depend on its fitness (Figure 3.7). That means on average the fittest will mate and survive into the next generation, but there is a non-neglectable chance that a weak individual will survive and one with a high fitness will not. This is a very important aspect as although some individual might be weak overall, it could contain some component that might turn to be useful in the following process.
There is an alternative version of the roulette wheel, called **stochastic acceptance**. An individual is chosen using the roulette wheel as described, but it will be selected only with a probability given by the ratio of its fitness to the maximal fitness value in the population. Otherwise, it is rejected and the selection continues.

![Figure 3.7: Roulette Wheel selection](image)

**Stochastic universal sampling** Being very similar to the Roulette Wheel, stochastic universal sampling is also a fitness proportionate selection method. It uses a single random value to sample the individuals by choosing them at evenly spaced intervals (Figure 3.8). That reduces the unfair nature of fitness-proportional methods, as it gives the weak individuals a chance to be chosen. Stochastic universal sampling might have bad performance if there is an individual with a fitness much larger than other members of the population. Then using non-evenly spaced intervals will help to decrease the effect of this individual on the candidate space.

![Figure 3.8: Stochastic universal sampling selection](image)

**Tournament** In this selection method, $k$ random individuals are selected. They are confronted with each other and the fittest is selected for mating. Selection pressure can be easily adjusted by changing the tournament size – for small $k$, weak individuals have a bigger chance to be selected. For $k = 1$, it is a purely random selection. The individuals already selected can either be removed from the parent population or kept there. In comparison with fitness proportionate selection such as the **Roulette Wheel**, tournament selection lacks stochastic noise. Among its advantages we can name that it allows selection pressure to be easily adjusted and works well on parallel architectures. Figure 3.9 shows an example for $k = 4$.

**Truncation** A specific fraction (a half / a third / . . .) of the best individuals is selected. This method is not much used in GAs, but it is actually a method for breeding animals and plants.
Random Unlike all the previous, we select any individual as parent with equal chance. It is also an equivalent to Tournament selection with only one competitor.

Concepts
A concept is a technique, that significantly affects the behaviour of a selection method but cannot work as a selection method by itself.

Boltzmann In many cases it might be useful to allow less fit individuals to breed more in the beginning of the run to maintain diversity of the population, but to later have stronger selection in order to promote highly fit individuals, assuming we are improving the solutions in the right part of the search space.

Boltzmann selection is a way to achieve this. It introduces a concept of temperature, which follows a fixed (decreasing) schedule and controls the rate of selection. Very common implementation is to set each individual’s fitness following this formula:

\[
Fitness(i,t) = \frac{e^{f(i)/T}}{\left[e^{f(i)/T}\right]_t}
\]

Fitness(i,t) is the fitness of an individual i at time t; T denotes the temperature and \([ ]_t\) is the average of the population at time t. We can see that when T decreases, the difference between high and low fitness values increases.

Reward-based This is a technique working well for a multi-objective optimization. The individuals can obtain rewards during their lifespan, which are awarded both to parents and newborns. The selection is then done proportionally to the accumulated reward. Reward-based selection can quickly identify the most fruitful directions of search by maximizing the cumulative reward of individuals.

Figure 3.9: Tournament selection with four competitors
Rank When using this concept, we order the individuals based on their fitness and then give them rank according to their position. The selection then depends not on the fitness, but only on the rank. In this method we do not need to scale the fitness function and it is also working with negative values. It is often used when the individuals have very similar fitness and would be selected with almost equal chance.

Non-random mating In this approach, we restrict the variety of individuals that can mate together. For example, we allow only individuals within certain distance of one another to mate. Then having a distance metric for individuals is necessary.

Elitism
Elitism is not a selection method by itself as it makes sense only for environmental selection, but neither a concept in the sense of those presented in this section. It describes the situation when not only the offspring, but also the fittest parents survive until the next generation. Unlike truncation selection on a combined population of parents and offspring, with elitism the parents are chosen independently. If, for example, all the offspring were better than the fittest parent, with elitism there would be still a pre-set percentage of parents surviving to next generation, while with truncation selection there would be only offspring.

3.4.5 Crossover
Crossover is the operator modelling biological recombination of single-chromosome (haploid) organisms. Its purpose is to recombine the parents into offspring. A well-designed crossover combines building blocks from both parents and thus improves the offspring.

1-point We select randomly one point in a 1-dimensional representation of an individual and then combine one part from each parent in the offspring, as illustrated in Figure 3.10a.

n-point We swap parts between n randomly chosen points. To maintain the building blocks hypothesis caution is required when choosing n not to break locally optimal parts of solutions. Figure 3.10b presents an example for n = 4.

Uniform For every gene, we toss a coin to choose which parent it will inherit from. The main difference from n-point crossover is that the uniform crossover enables the parent chromosomes to contribute not with segments, but directly with genes. That might be useful for some tasks, where such behaviour is desired (such as knapsack) and very contra-productive in others (Travelling Salesman Problem). Since it does not maintain longer schemas, it is more exploratory than the n-point crossover and therefore a more complete search of the space is conducted. Figure 3.11a illustrates this method.

Arithmetic Used for example in floating point or integer representation. The values of parents are combined during the crossover — most often the
(weighted) average (see Figure 3.11b) of parents is used with the following formula:

\[ \text{Offspring} = ax + (1 - a)y \]

It can be combined with some of the structural crossovers above.

**Multiple parents** There is no reason to limit ourselves with biological concepts and have only one or two parents. As usually, the usage differs with representation and operators. For example in real value representation, we can use a combination of the parents’ values to calculate the value of the offspring. In binary representation, we can use in the offspring for each bit the value a majority of the parents agrees on as in Figure 3.12.

**Ordered Chromosomes** Clearly, the above mentioned crossovers can hardly be used for chromosomes where the order of the genes matters, or where each value can be used at most / only once. A good example is an ordered list of cities a travelling salesman has to visit. Therefore there are several crossovers designed specifically for ordered chromosomes. We will not go into detail as they are irrelevant for our task.
3.4.6 Mutation

If we had only crossover, it would be hard to introduce new concepts within building blocks. Therefore we use mutation, which typically changes the chromosome locally. They are usually designed to prevent getting stuck in local optima. The new value for the mutated gene can be either chosen from the whole domain, or we can get it by shifting the current one. We do not mutate all individuals in a population, but choose them randomly according to some pre-set probability.

**Bit Flip** With a probability $p_m$ it changes a value of each bit ($p_m$ very small to change in average just one bit per individual)

**Flip All Bits** All the bits of the genome flip their values.

**Scramble** A mutation popular with permutations. It takes a part of the individual and randomly repositions all the values.

**Inversion** In this mutation, we take a block in the individual and reverse it.

**Boundary** A mutation for integers and floats. It randomly replaces the genome with either its lower or upper bound.

**Uniform** A mutation for integers and floats. A uniformly selected random value between specified lower and upper bounds is used to replace the value of a chosen gene.
**Non-Uniform** A mutation for integers and floats. The mutation rate decreases during the evolution.

**Gaussian** A mutation for integers and floats that adds a unit Gaussian distributed random value to the chosen gene. It has to fall within specified bounds.

**Problem specific** As we have already mentioned, it is highly recommended to use problem specific operators. In case of mutations, that usually means a mutation locally optimizing the individual based on domain specific information. A good example is the 2-opt mutation (Figure 3.14), which, on an ordered integer representation of a trip between multiple cities, locally improves the order of cities by checking if swapping two adjacent cities improves the path or not.

![Figure 3.14: Example of a 2-opt mutation. The crossed out edges are changed for the red ones.](image)

**3.4.7 Combining Operators**

Some operators help with variation, some repair errors in solutions and others direct the search. Having only a selector would result into ever-increasing number of the best individual from the initial population, selection and crossover will likely make it to converge to a local optimum and using mutation alone leads to a random walk (random mutation) or very quick convergence (repairing). Therefore, all of them improve the solution individually, but they have to join forces in order for the algorithm to succeed in the task. Not only that, when mixing multiple operators, it should be considered whether they complement each other well and do not destroy one another’s work.

As an illustration, we can see in Figure 3.15 how crossing over two individuals moves their offspring to a different position in the fitness landscape. Using only mutation would not help with getting over the deep valleys of the function.

**3.4.8 Exploration vs. Exploitation**

There are two core concepts for guiding the search in GA – *exploration* and *exploitation*. The first refers to exploring the search space looking for promising areas, while the second refers to focusing on improving an already known solution. Using only one of them does not work well. If we are only exploring, we are doing a random walk and not using any knowledge we gained during the computation.
On the other hand, using only exploitation leads to getting stuck in local optima and to rigidity.

Therefore both concepts need to be combined to achieve better results. Usually we start by exploring the space and then move on to exploit the knowledge we gained. Practically, this can be done for instance by changing probabilities of some operators – crossovers are mostly considered to be exploring and mutations to be exploiting.

### 3.4.9 Termination

The whole evaluation is typically terminated when one of the following events occurs:

- A set number of generations is calculated.
- There has been no improvement for some set number of generations.
- We have reached some set value.

### 3.5 Lifetime Models

So far we spoke only about the Darwinian model of evolution. This is based on the concept that only individual’s genotype can be transmitted from parent to an offspring by means of natural selection and genetic variation. There are, however, other models and two of them will be presented in this section. They are used when we hybridize a GA with local search – then we can use either Lamarckian or Baldwinian model to deal with the individuals produced by the local search. There is an ongoing discussion which of the models is the best; again, the choice is very problem-dependent.
3.5.1 Lamarck

Lamarck says in [52] that traits and features acquired during one’s life can be passed to children. This is a concept ignored by natural biology, but from a computational point of view (see [53]), it gives good results to some problems. We simply search locally in the neighbourhood of an individual and if a better solution is found, it becomes the offspring.

3.5.2 Baldwin

In the Baldwinian model [54, 55], unlike the Lamarckian, we do not transmit the acquired traits to the offspring, but neither we ignore it like in the Darwinian model; the individuals can have a tendency to learn beneficial behaviours. This is achieved by locally searching the neighbourhood and if a better solution is found, we assign to the individual not the trait itself, but only the better fitness. This enhances the individual’s chances to survive and eventually acquire this trait later by means of Darwinian evolution.

3.6 Knapsack

This section enhances the understanding of the GA theory presented above by applying it to a well-known example problem. The Knapsack problem is a well known combinatorial optimization problem. The task is to maximize the value of objects in a knapsack without exceeding its capacity. It is interesting from several perspectives:

- The decision version\(^1\) is NP-complete.
- The optimization problem\(^2\) is NP-hard.
- There is a pseudo-polynomial time algorithm that solves it in \(O(n^3)\) using dynamic programming.
- There is a polynomial approximation scheme.

Therefore one of the research interests lies in identifying the difficult instances of this problem as in [50, 57]. We are, however, more interested in solving this problem using a genetic algorithms, which is described in [29, 30, 31, 32]. Genetic algorithms provides a way to solve some instances of the knapsack problem in linear time complexity.

Task

We are given a set of items, each with a weight and a value. The task is to determine the number of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

\(^1\)Can we achieve particular value without exceeding the capacity of the knapsack?  
\(^2\)What is the best value we can achieve?
Solution

For this task, a good encoding is to represent the individual as a binary vector of length equal to the number of items, with 1 denoting the corresponding item being present in the knapsack and 0 being the opposite. The fitness is then simply the sum of values of items present in the knapsack. We can easily use bit flipping as a mutation and either allow only mutations not exceeding the item limit or penalize those exceeding it when calculating the fitness value.

As far as crossover is concerned, \( n \)-point crossover (typically \( n = 1 \) or \( n = 2 \)) makes good sense. It means that combinations of items from both parents are mixed in the offspring.

Because we have no knowledge about how good solution looks like and the operators do not restrict the search depending on the initial population, we can initialize it randomly.

It could seem like a good idea to use integer representation of values of the items, and zeros whenever an item is not present, as that gives more information and the fitness of the individual could be computed simply as the sum of all numbers present in the individual. In that case we could use the flip mutation like with binary representation, only flipping the actual non-zero value of the item and 0. The problem comes with crossover – we cannot use \( n \)-point crossover for any \( n \), since the items have different values and the crossover would, in most cases, produce invalid individuals.
4. Designing a Genetic Algorithm

After having presented GAs in the general sense in the previous chapter, we proceed to apply them to a specific problem. This chapter focuses on the design of GA mechanisms for our particular task, while the next one explains the actual implementation.

We begin by a statement of our goal, followed by explaining our expectations of the results. Then we move on to parts of the GA — initialization, crossover, mutation, selection and fitness. In the end, we discuss edge-disjoint subgraphs.

4.1 Goal

We want to design a GA that would find solutions to some of the hypercube problems presented in Chapters 1 and 2.

We will mostly focus on the search for 3-spanners with minimal number of edges in hypercubes of small dimensions, since of the mentioned problem it is the most appropriate for a GA, because the fitness function is rather straightforward and a good global solution is typically composed of good local solutions.

Once the GA is implemented, we plan to try to find solutions to other hypercube problems, namely searching for a spanner with minimal maximal degree and finding multiple edge-disjoint spanners or spanning trees.

In this chapter and the following chapters, we say that an edge is activated if it is both part of the hypercube and also of the subgraph represented by an individual. The size of the spanner is the number of activated edges.

4.2 Expectations

We are fully aware of the computational complexity of most hypercube-related problem. The exponential growth of the hypercube with each increase in dimension means that solving any high-dimensional problem computationally is impossible. Nevertheless, even for small dimensions, the results we could get would be interesting and useful. If we managed to obtain a subgraph with size lower than the known upper bound, we would prove this bound to be suboptimal.

We have already mentioned several times that GAs are not suitable for all problems, so an apparent question comes up: Are they suitable for hypercubes? If we look at spanners and consider the hypercube in terms of the Schema Theorem (Section 3.3), the answer is likely to be positive, since locally optimal spanners are short objects of low degree and thus should spread across all the population. Concerning other properties such as minimizing maximal degree, the combination of locally optimal solutions does not necessarily lead to global optimum. But since for this task we only need to slightly modify the algorithm, it is worth a try.

Even if we fail to get a 3-spanner better than the upper bound, it will be interesting to see how close we can get to the bounds. Moreover, if the results make it plausible, we could combine different properties and try to search for instance of a 3-spanner with more edges than optimal, but with a low maximal degree as well.
4.3 Representation

There was only little doubt about what representation we should choose for the hypercube subgraphs. Since we can naturally translate the problem of edges being part of the subgraph into one binary value for each edge, and we know the number of edges is the same for each hypercube of a set dimension, using binary representation of length equal to the number of edges in a hypercube of given dimension is clearly the best solution.

However, we have to realize that we cannot transform the information about the edges into a string while maintaining any kind of adjacency from the graph in the string. That is, if two edges share a vertex, they need not be within proximity of one another in the string. It is not an obstacle, but we need to be aware of this when designing the operators and trying not to break locally good solutions.

Seeing that we are talking about spanners, it might seem that using trees for representation would be more fitting. We have to note that a spanner is not necessarily a tree — actually, it is rarely a tree — and therefore trees would not be very suitable representation.

4.4 Initialization

Since it is hard to predict the GAs behaviour without trying, it only makes sense to implement all the initialization options and experiment with them.

4.4.1 Random

We let the user to set the probability that an edge will be activated in an individual. This, on average, sets the percentage of active edges in the spanner. This setting therefore affects if we will try to aim for the perfect size and only move the active edges around, start with too small subgraphs and try to repair them by adding new edges or start with an almost complete hypercube and remove edges when looking for the optimum.

4.4.2 Good Spanner

It is possible to initialize an individual using a description of a spanner that we either obtained during previous runs or that we constructed separately based on some known theoretical construction, such as in Section 2.3.

4.4.3 Good Spanner on a Subcube

As we just mentioned, in Section 2.3 we have learned about good constructions for small dimensions. We can use them not only to initialize the whole individual, but also to initialize subcubes within an individual of higher dimension. However, since the constructions we know cover mostly the dimensions we are able to reasonably compute, this approach stays as an idea for further consideration, but is not implemented.
4.5 Crossover

Because we are working with binary representation, we can use some of the classical crossovers. In addition, we should try to exploit hypercube specifics to get better results.

4.5.1 \( n \)-point

We’ve already familiarized ourselves with this crossover in Section 3.4.5. Applying it to a hypercube represented as described in Section 3.4.2 we have to bear in mind that the building blocks within the representation are not linear. Therefore it is simply a lottery if the chosen \( n \)-points will break our good local subgraphs or not. A good value for \( n \) will have to be found in an experimental way.

4.5.2 Subcube Swap

The regularity of a hypercube calls for a crossover that would honour its structure. Swapping subcubes – hypercubes of smaller dimension – is a perfect way to do that. We can combine two parents and produce two offspring as shown in Figure 4.1.

The size of the swapped hypercube will greatly affect the overall behaviour of this crossover. The minimal subcube worth swapping is of dimension two and for hypercube of dimension \( n \), swapping subcubes of dimension bigger than \( n - 1 \) clearly makes no sense.

![Figure 4.1: Subcube Swap Example](image)

4.6 Mutations

With crossover we focused on combining large parts of individuals. Therefore our mutations should be designed as complements and make mostly only local changes on one or a few edges.

4.6.1 Random

The simplest and very common mutation for any binary representation is to randomly flip a bit. Usually two probabilities are set: one to decide if an individual
will get a chance to mutate and the second choosing for each bit if it will flip or not. Such mutation may help but only rarely solves issues with local optima.

This mutation makes good sense in our task. It corresponds to randomly flipping the activation status of an edge. Supposing we’re using subcube swap as a crossover, the mutation should complement it by adding random changes within such subcubes.

4.6.2 Repairing

The idea behind this mutation is that it will repair the subgraph to have the desired property. For 3-spanners, it means adding an edge to a subgraph containing an inactive non-detoured edge $e$ in order to maintain the 3-spanner property for $e$. The simplest method is to activate an inactive edge that does not have an active detour. However, we need to make sure that probability of this mutation is reasonably low, because adding too many unnecessary edges in a single step would not serve the purpose.

Using this mutation makes the algorithm to converge to a 3-spanner quite early; on the other hand, as no invalid subgraphs survive for more than few generations, it is hard to improve this early local optimum. Therefore the usage of this mutation needs to be careful and well motivated.

A straightforward optimization of this mutation is trying to compute the optimal edge to activate — not just simply the missing one, but possibly any of the inactive edges in its detours — to minimize the number of activations needed to achieve the 3-spanner property. This is considerably slower to compute, but seems to be worth it because it converges even more effectively.

4.6.3 Improving

Another approach is inspired by the 2-opt mutation mentioned in Section 3.4.6, which optimizes the graph locally. However, such approach requires us to know a specific property of the graph that would allow us to improve the properties of the subgraph by simple local transformation.

Thanks to Section 2.3 we know how to form subgraphs of decent properties, but it is yet to be discovered if those are truly optimal or would only result in the GA getting stuck in a local optima – the latter being our opinion on the subject.

Since we failed to discover a generally valid property distinguishing a better subcube from a worse one, we could not implement any improving mutation.

4.6.4 Translation and rotation

With swapping subcubes being the crossover, it makes good sense to also rotate the subcubes. We have stated in Section 1.2 that any automorphism can be achieved via a combination of rotations and translations and the order does not matter. Since we can let those happen accumulatively over several generations, we need only one translation and one rotation as the operator; or a combination of the two with a pre-set probability to perform either or both.
4.7 Selectors

Without knowing more about the fitness function, it is hard to tell which selection method will be optimal. Using the most common, such as **Roulette Wheel** and **Tournament** seems to be the best choice. In case we want to control the selection pressure, we can also implement $k$-tournament. By choosing the right value for $k$, we have little more control over selection pressure compared to the roulette wheel.

Because great variety is needed in the beginning and rather fine refinements towards the end, we extend our possibilities by adding **Boltzmann** selection to our mix. We can combine it with both the roulette wheel and the tournament and see if it brings any interesting results.

**Temperature**

Boltzmann selection requires a function for calculating the current temperature $T$ at any given time $t$. The following formula was suggested in [34]:

$$T_t = T_0 \cdot (1 - \alpha)^k$$

$$k = 1 + 100 \cdot \frac{t}{MaxGen}$$

$MaxGen$ is the maximum number of generations, $T_0 \in [5, 100]$ denotes the starting temperature and $\alpha \in [0, 1]$.

**Replacement**

Replacement is the only relevant aspect of GA that was not mentioned in Chapter 3. It is the process of creating the selection pool for the environmental selection, which selects the members of the new generation. For the hypercube task, the following replacement strategies were included:

- **ChildrenOnly** As the name suggests, only offspring are considered for the new generation. It is the fastest method, because it requires no further computation.

- **Mergeing** The parents and the offspring are merged into one population from which the next generation is selected.

- **Percentage** A given percentage $p$ of the best parents and a $1 - p$ of the offspring are considered for the new generation. This is also a kind of an elitist procedure.

- **Elitism**

Clearly, we do not want to lose the best individuals due to some unlucky mutations. Also, we believe that at some point, a high-scoring individual can be created by removing an edge but adding it some place else a few generations later. But in order for that to be possible, this individual must still be present in the population. Therefore we add elitism to make sure our top solutions are always available to minor changes like this.
4.8 Fitness

There is hardly a good GA without a fitness function that guides the search in the right direction. Since we aim for multiple directions, we will explore them one by one.

4.8.1 3-spanner size

When searching for a 3-spanner with as few edges as possible, we are typically interested in the number of edges $edgeCount$ and the ratio of detoured to the sum of detoured and non-detoured edges $spanner$. Since GA is trying to maximize the fitness, we can either use the value of $1/fitness$ for arbitrary fitness, or specifically for our task $|V(Q_n)| - edgeCount$; that is, we are trying to maximize the number of edges not present in the subgraph.

Some of our ideas for a good fitness follow.

- $|V(Q_n)| - edgeCount$  
  The simplest solution that does not reflect the 3-spanner property at all. Therefore clearly nothing that could work. This difference will be further denoted as $missing$.

- $missing - missing * 3-spanner$  
  Improving the previous fitness, we now aim for 3-spanners. The value of $3-spanner$ should be 1 when the subgraph is not a 3-spanner. Therefore all non-3-spanners will have fitness equal to zero and thus be likely to die quickly. This does not reflect how close they are to becoming a valid 3-spanner, which greatly affects the selection.

- $missing * spanner$  
  Dealing with the problem from the binary version, the $spanner$ is the ratio of detoured to the sum of detoured and non-detoured edges — sort of a percentage of 3-spanner quality. Using this for fitness function, we can distinguish non-spanners that are close to becoming spanners and thus favour them during selection.

4.8.2 Maximal degree

Trying to minimize the maximal degree, we have to note that the values themselves are so low we cannot really use them directly as fitness. We should also reflect how many vertices with the highest degree are in the subgraph. It seems logical that a subgraph with four vertices of the highest degree is closer to lowering its maximal degree than a subgraph with ten of them. Therefore the basic idea is to apply some function to the degrees and then use it in the denominator to maximize instead of minimizing.

- We can sum degrees over all vertices and then calculate average. Then, however, two vertices of degree two give the same value as one of degree one and one of degree three, which is clearly not desired.

- Next idea is to use a higher power of each degree to make the higher degrees weight more. This approach works considerably better but once the initial variation is gone, it does not improve much.
• Another approach would be to get inspired by multi-criteria optimization.

For minimizing the maximal degree, there is no clearly best choice for a fitness function.

4.9 Edge-disjoint subgraphs

In this section, we firstly explain why using the 3-spanner property within our program cannot produce any viable outcome regarding edge-disjoint spanners. Then we offer a quick theoretical insight into how it can be used to get spanners with longer detours.

4.9.1 Edge-disjoint 3-spanners

We know from 1.3 that a 3-spanner must have at least $3 \cdot 2^n$ edges, while a hypercube has $n \cdot 2^{n-1}$ edges. Therefore even with the lower bound being actually achievable, we could construct multiple edge-disjoint spanners only beginning with dimension 12. However, so far for $n = 12$ we only managed to compute a 3-spanner with about 60% edges extra (see Table 7) to the value of the upper bound. Also, at this dimension the speed of the program could be described as “fairly slow”. Therefore since we would need to use even higher dimension, it makes little sense to do so.

4.9.2 EDS with longer detours

By Lemma 19, we know that the lowest dimension where trying to find a delayed subgraph makes sense is for $k = 2$. Therefore $d = 4k - 2 = 6$. However, with the not-so-promising results that will be shown in Chapter 7, we decided not to implement additional delay checks. Therefore we could not test this property, but as described in Chapters 5 and 6 any user with programming knowledge should be able to add/modify the code to use $k$-delay as the main property for edge-disjoint subgraphs.
5. CubeBreeder

This chapter describes the implementation of the evolutionary algorithm analysed in Chapter 4. We begin by describing the data structures and continue with the algorithms. User documentation follows in Chapter 6.

Programming language

Since performance is dependent more on the dimension of the problem rather than implementation details, the choice of a programming language was not necessarily limited to some of high efficiency. We decided for C# as there were no clear disadvantages and we felt most comfortable using it. A great book about this programming language is [58].

A question might arise why we develop a tool of our own rather than using one of the many open-source libraries designed for GA. The reason is that we believe the hypercubes and the corresponding operators are so specific and the general concept of GA is so straightforward that it seemed much easier to write everything ourselves. We have to note that the overall design of the implementation and some naming conventions are loosely based on GitHub repository https://github.com/martinpilat/evaTeaching, which is written in Java.

5.1 Data Structures

In this section, we briefly analyse the properties of a good hypercube GA representation and then move on to describe how we implemented it.

5.1.1 Analysis

A good representation is the core of a good solution, especially since the number of vertices in a hypercube grows exponentially with its dimension $n$. And because every vertex has exactly $n$ edges, we cannot compress the representation by omitting any of them. Therefore the space complexity will be at least $O(2^n)$. On the other hand, this allows us to set some bounds that will hardly be exceeded with current computational capacity. For example, we can use four byte integers as indices for vertices, because we will not need to run it for dimensions higher than $n = 31$, where it would go out of bounds.

As already mentioned in Chapter 3, we assume that the most common operation will be calculating the fitness value. That means, for most of fitness functions proposed in Section 4.8, checking whether a newly created hypercube has the desired property $^1$ e.g. is a 3-spanner. We decided to design our data structures accordingly and optimise in this manner by pre-calculating useful and frequently used pieces of information rather than obtaining them on the spot. Unfortunately, in a dynamic environment of changing individuals, not all can be pre-calculated.

We can pre-calculate all possible local detours for every edge and build an easily navigable data structure that would allow us to access them efficiently. In

---

1That turned out to be the right assumption during code profiling.
this way checking if a non-active edge has a detour requires nothing but a simple run through a list of its possible detours while checking whether the edges within them are active. Memory consumption of this approach is not too high, since we only reference objects that represent edges — more on that in the next section.

## 5.1.2 Representation

The general concept is as follows:

- **Edge** – An object that has information about its vertices, the detours that use it as an intermediate step and the detours that can be taken if it is not active.

- **Hypercube Graph** – There is only one for the whole program and it contains all the hypercube-related information that is the same for all the individuals. Basically, it describes the whole hypercube structure for a full hypercube of the desired dimension.

- **Individual** – A subgraph of the Hypercube Graph. Carries the information which of the edges in the full hypercube are in the subgraph. Also has some data related to the GA.

Let us now present the above mentioned in a more detailed way.

**Edge**

An Edge class presented in Figure 5.1 has two vertices Vertex1 and Vertex2 that represent its endpoints. The vertices can be represented by integers — with 64bit integers we can code hypercubes up to dimension 64. Since that is way beyond our computational ability, we will use only four byte integer type `Int32`.

Every edge has a unique ID. These IDs are numbers from 0 to $n \cdot 2^n - 1$ where $n$ is the dimension of the hypercube. This allows us to use IDs as indices into an array representing the activity of edges within an Individual. Note that similar IDs do not imply proximity of the edges.

Further, each Edge carries a list of triples of `int`, which correspond to the IDs of edges this Edge can use as a detour. The main advantage we see in this implementation is that the detours can be easily accessed. We already mentioned that checking the detours is computationally the most demanding part, so making it as efficient as possible is desired.

```csharp
class Edge
{
    int ID;
    int Vertex1;
    int Vertex2;
    List<Tuple<int, int, int>> detourMe;
}
```

**Figure 5.1**: Summary of the Edge class.
Hypercube Graph

The hypercube graph is represented by the GraphInfo class, which is the most complicated object. It contains all the information about edges and detours. As shown in Figure 5.2, the GraphInfo carries the information about its dimension, a two-dimensional arrays of Edges called graph and another one-dimensional array of Edges.

The array graph can be interpreted as an $v \times n$ matrix where $v$ is the number of vertices and $n$ is the dimension. The idea is that each vertex has its line and each direction in the binary representation of the hypercube has its column. For example with dimension $= 3$, we find at graph[3][1] the edge from vertex number 4 to vertex number 6, since 100 and 110 differ in the direction at index 1.

The edges array has a simple purpose of providing access to all the edges based on their ID. This is a frequent operation and therefore we thought it is useful to make it easier. Obviously, all the edges in the arrays are only references to an Edge object.

Within the program we keep only one instance of the GraphInfo object for all individuals to share and to find any graph-related information they need.

```csharp
class GraphInfo
{
    int dimension;
    Edge [][] graph;
    Edge [] edges;
}
```

Figure 5.2: Summary of the GraphInfo class.

Individual

The Individual shown in Figure 5.3 represents a solution to our problem. It has an objectiveValue to provide the user with feedback about its quality and the fitnessValue to be a valid part of the GA. The chromosome is implemented as an array of bytes, where an edge is active if the value at the corresponding index is greater than zero. Note that a boolean type, which would be sufficient for a binary representation of chromosomes, takes in C# one byte of memory as well. Therefore the usage of byte requires no extra resources and we can elegantly use it later when colouring edges for edge-disjoint spanners.

```csharp
class Individual
{
    double fitnessValue;
    double objectiveValue;
    byte [] edgeActivity;
}
```

Figure 5.3: Summary of the Individual class.

\^2We are indexing from 0.
5.2 GA Implementation

For most of the operators we use their implementation is not worth mentioning. For instance selectors are quite simple and therefore it makes little sense to describe them in detail. Thus we will focus only on some of the more interesting ones.

5.2.1 Subcube Swap Crossover

Since an exchange of promising subcubes is highly desired, the crossover should be designed to keep that in mind. With integer representation of vertices, we can achieve this by fixing certain $k$ bits and exchanging the rest. By fixing some bits, we get three kinds of edges in each of the two parents as seen in Figure 5.4 – the “outer” layer outside of the affected subcube, in which no edges are swapped, the “inner” layer where all the edges are swapped and the edges connecting these two layers. For the layer-connecting edges, in the new individual we activate the edge if it is active in at least one of the parents.

![Figure 5.4: Illustration of the layers of the Subcube Swap Crossover with fixed bits at indices 2 and 3 with fixed value 0. The “outer” layer is in blue, the “inner” layer in green and the connecting edges in yellow.](image)

5.2.2 Edge-Repairing Mutation

When repairing non-spanner subgraphs, we use two types of mutations — simple and clever (for details see Section 4.6.2). While the simple one for each non-detoured edge based on some probability decides to activate it or not, the clever one, whose pseudo-code is in Figure 5.5, is more complex.
Input: A representation of a hypercube subgraph $G$

Initialization: $\text{max} = 0$, $\text{counter} = 0$;

$\text{repairs} = \text{new Dictionary}\langle\text{Edge, int}\rangle$;

foreach edge in non-detoured edges in $G$ do

    edge $\leftarrow$ active;
    count = GetNonDetouredEdges();
    repairs.add(edge, count);
    if count > max then
        max = count;
        counter = 0;
    else if count == max then
        counter++;
    edge $\leftarrow$ not active;
end

foreach edge in non-detoured edges in $G$ do

    if (Random(0,1) $< 1 / \text{counter}$ and repairs[edge] == max then
        edge $\leftarrow$ active;
    end
end

Figure 5.5: The core of the algorithm intelligently repairing an invalid 3-spanner.

The idea is to try to find one edge within all the non-detoured ones whose activation would solve as many detours as possible. Clearly, each of them will reduce the number of non-detoured edges at least by 1, however, some of them will also fill a missing part in detours of other non-active edges and thus reduce this count even further. To find out, we simply try to activate each of them and recalculate the number of non-detoured edges in such subgraph. We keep a counter of the number of occurrences of the current maximum.

Finally, we activate each of the edges solving the maximal number of issues with probability of $1 / \text{counter}$. Therefore on average just one of them should become activated.

5.2.3 3-spanner Property

The check whether a subgraph represented by an individual is a 3-spanner is fairly straightforward. At the same time it is computationally the most demanding part.

As we see in Figure 5.6, three counters are needed for the output:

Active – The edge is active.

Detoured – The edge is not active, but has an active detour.

NonDetoured – The edge is neither active nor has it an active detour.

The algorithm iterates over all edges, active or not active, within the individual. For each of those edges it checks whether it is active or not. If it is active, it increments the active counter. If not, it checks the list of possible detours one by one and if all the edges in any of them are active, it increments the detoured counter. If none of the detours were active, the nonDetoured counter is incremented.
**Input:** A representation of a hypercube subgraph $G$

**Output:** The percentage of the subgraph being a 3-spanner

**Initialization:** detoured, nonDetoured, active = 0;

```plaintext
for $i \leftarrow 0$ to number of all edges in $G$ do
  |  if edge $i$ is active in $G$ then
  |     active++;
  |  else if edge $i$ is detoured in $G$ then
  |     detoured++;
  |  else
  |     nonDetoured++;
  | end

if detoured + nonDetoured == 0 then
  |  return 1.0;
else
  |  return detoured / (detoured + nonDetoured);
end
```

**Figure 5.6:** The algorithm to check if a subgraph is a 3-spanner

Once all edges are checked, the return value is calculated. It is simply the ratio of detoured edges to non-detoured in the subgraph. That is, if the subgraph is a 3-spanner, the algorithm returns 1.

### 5.2.4 Spanning tree

There are two uses for a check whether a subgraph is a spanning tree or not. First is when we want to favour spanners with more edges over non-spanners with fewer edges in the GA. Then we can simply return 0 in the 3-spanner check if the subgraph is not a spanning tree. That obviously slows down the computation and the benefits do not seem to be worth that. Second is to check the spanning tree property when calculating edge-disjoint spanners. Then the basic option — and the only one the program is capable of at the moment — is to check for edge-disjoint spanning trees.

To investigate this property we use a well known approach. We count the components of the subgraph and if there is only one, we know that the subgraph is a spanning tree. The algorithm in Figure 5.7 iterates over all vertices and for each that was not yet visited (that is its state is 0) it runs a depth first search (DFS – Figure 5.8) for other edges in the same component. The counter counts the components.

In a spanning tree, starting with an arbitrary node, a single DFS run will set all the vertices in the graph to the finished state. We could terminate the counting if we come across a second unfinished vertex after already calling the DFS. Such method is called **IsSpanningTree** and is part of the program. We, however, prefer to lose some time but get the information on the number of components, because we can use it in some fitness function.
Input: A representation of a hypercube subgraph G
Output: The number of connected components in G
Initialization: new state[number of vertices];
counter = 0;
for \( i \leftarrow 0 \) to number of vertices in \( G \) do
  if state[\( i \)] == 0 then
    counter++;
    DFS(i, state);
end
return counter;

**Figure 5.7:** The algorithm to count components in the subgraph

Input: int vertexIndex, int[] state
state[vertexIndex] = open;
foreach edge in all edges in vertex vertexIndex do
  if edge is active then
    k ← the second vertex of edge;
    if state[k] is unopened then
      DFS(k, state);
    end
  end
state[vertexIndex] ← closed;

**Figure 5.8:** DFS Colouring algorithm

### 5.2.5 Parallelism

Since the task is very resource-consuming already for dimensions around \( n = 10 \), some optimizations were needed for better performance.

In a GA as described in Chapters 4 and 5, there are many actions that can be done simultaneously. Also, in a time-consuming calculation it is highly useful to do as many things as possible in parallel. However, most of the actions performed during the run of a GA are very quick in their nature. Also, parallelizing for instance crossover and mutation cannot be done, as the order for them mostly matters. Therefore utilization of parallelism within a single run of a GA cannot be considered useful.

In order to obtain credible results, the GA needs to be run multiple times with different seeds for its random numbers generator. Clearly, parallelizing the whole runs (represented by the Run object) would make the computation considerably more effective. However, when running in parallel, each instance of the GA needs to keep its own generator in order not to affect each other. We added independent random numbers generators and log files to store data to every individual Run object. Only the first Run is allowed to use the console for output, since mixing outputs from multiple runs, even labelled, would be highly confusing.
5.3 Input / Output

Except for the configuration of the GA, only two entities are possibly needed as inputs. Coincidentally, they also happen to be outputs of the program.

One of them is the pre-calculated hypercube structure in GraphInfo and the other is a description of a subgraph — basically the Individual.

5.3.1 Hypercube Structure

Clearly, the hypercube structure for each dimension never changes. Therefore, to save time on multiple runs, after creating the graph structure as described in 5.1.2 the program saves it on the hard-drive and on subsequent runs checks for its presence and if present, loads it.

Since the object is not intended for any external manipulation, it is saved as a serialized object written in binary format. These files get considerably large for higher dimensions – reaching over 100MB for \( n = 14 \) — they are compressed before being saved. That reduces their size more than four times. 8

5.3.2 Hypercube Subgraph Description

A description of a subgraph with the desired property is the main outcome of the program. It is the best individual the GA is searching for. We designed a simple, human readable format for handling this task – an example of which can be seen in Figure 5.9. It begins with a summary of the subgraph containing information about the total number of edges and the maximal degree, followed by a list of edges present in the subgraph — i.e. active in the Individual. They go one per line and are first described by the vertices converted to decimal using big endian and then by their binary representation. When running an edge-disjoint fitness function that requires edges of multiple types, there is a fifth column at the right with integers denoting the colour of the corresponding edge in the row.

<table>
<thead>
<tr>
<th>Total of edges with max degree 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Figure 5.9: Input / Output formatting of the subgraph of \( Q_3 \). Note the order of appearance of the edges.

As mentioned in Section 4.3 it could be useful to initialize an individual according to some pre-calculated subgraph. Therefore a method for transforming the output of CubeBreeder into an Individual was implemented so that the
output can be used again for initialization. Moreover, in Section 2.3, some good small dimension constructions were shown. In order to be able to initialize an individual using such construction, a python script called spanner-generator.py (see Attachment) was written. It creates an input in the format presented in Figure 5.9.
6. User Documentation

This program is by no means designed for a general user. It is a tool for running a particular GA. Therefore rather than explaining how to run it without any deeper knowledge of the task, we will attempt in the user documentation to show a researcher how to use it, understand it, modify it and possibly extend it.

Having this concept and goal in mind, we thought unnecessary to implement checks on formatting of input or user friendly handling of the controls. That by no means implies that the program might crash at any point, but rather that user should follow the guidelines described in this chapter because he will not be warned that there might be any issues during runtime.

We recommend to the potential user to download the current version from GitHub repository [https://github.com/Hekit/CubeBreeder](https://github.com/Hekit/CubeBreeder).

6.1 What is configurable

The user can set the size of the population, dimension of the hypercube, the number of generations, the size of an elite, the rate of output on console, number of repeats, whether to run in parallel and the output Directory for calculated spanners and logs.

The initialization can be configured in terms of the percentage of edges to be activated in an individual, usage of some already calculated subgraphs and the ratio of these two in the population.

Default values for probabilities can be set, including those for mutation, crossover and mutation per item. Other configurable probabilities include winning chance of weaker competitor in tournament and changing subcube size.

Selectors can be specified both for environmental and mating selection. Roulette Wheel, Tournament and Boltzmann variant for both the previous selection methods are available. ChildrenOnly, Merging and Percentage replacements can be used during the environmental selection process.

There are multiple operators that can be arbitrarily combined and their specific properties can be set. Not only that, each operator’s properties can be automatically updated once every generation. For crossover, we have SubcubeSwap with specifiable probability and the size of the subcube and N-Point with specifiable probability and number of points. For mutations, we have Clever Repair of an edge and Simple Repair for an edge. In both, the probability of their occurrence and the probability of them modifying each edge can be set. The same properties can be set also for the Flip Edge mutation. Translation and Rotation can have probability and the size of the subcube specifiable.

The fitness can be set to optimizing for 3-spanners, minimal maximal degree or edge-disjoint spanning trees. The details of these fitness functions can also be modified, but only directly in the code.
6.2 How to configure

The most user-friendly way to configure most of the features is via a configuration file, as described in this section. A little impractically, this file needs to be named config.txt and has to be placed in the same directory as the executable of CubeBreeder. Note that not all features can be configured in this way. For some, the code has to be modified directly.

In case the user would be interested in changing the default values, that can be done in the Settings part of the project’s Properties or, not so elegantly, in the Settings class.

Now let us explain the specifics of the configuration file. However, before we begin with notation, let us accentuate that the order of sections matter. The order of lines within a section matters not. If parts of instructions are omitted the program will use default values.

Notation

In this section, we will use the following notation when describing the lines of the configuration file.

@I Integer value.

@D Real number.

@Path A path to a directory.

A | B Either A or B.

[A] A is an optional parameter.

[A [B]] B is an optional parameter but can be used only when A is used.

General settings

PopulationSize @I Sets @I to be the size of the population.

Dimension @I Sets @I to be the dimension of the hypercube.

Generations @I Sets the upper limit for the number of generations to @I.

EliteSize @D Sets the elite size ratio to @D.

OutputRate @I Sets the terminal output to every @I generations.

Repeats @I There will be @I runs.

Parallel If used the runs will be parallelized.

OutputDir @Path aa
Initialization

InputFile @Path
Sets the path to the initialization file, which needs to bear the name n_init.txt, where n is the intended dimension. Unless FileUsage is specified, the default value for file to random initialization ratio will be used.

Percentage @I
Sets the percentage of edges to be activated during random initialization.

FileUsage @D
Sets the ratio of individuals initialized using a file. If no file is provided, this will have no effect.

Probabilities

Mutation @D
Sets @D as the default mutation value.

Crossover @D
Sets @D as the default crossover value.

MutationPerItem @D
Sets @D as the default value for mutation of a single item.

TournamentWeaker @D
Sets @D as the probability the weaker competitor will win.

ChangingSubcube @D
Sets @D as the probability the subcube size will change with every update.

Selection

We strongly recommend using only one of the following per each type of selection - environmental (ENV) and mating (MAT). If neither of the parameters is set, the selection method will be used for both.

RouletteWheel [ENV | MAT]
Sets the Roulette Wheel as selector.

Tournament [@I [ENV | MAT]]
Sets the Tournament as selector with @I as the number of competitors.

BoltzmannRouletteWheel [ENV | MAT]
Sets the Boltzmann Roulette Wheel as selector.

BoltzmannTournament [@I [ENV | MAT]]
Sets the Boltzmann Tournament as selector with @I as the number of competitors.

ChildrenOnly
Sets ChildrenOnly as the replacement.

Merging
Uses Merging replacement.

Percentage @I
Uses Percentage as the replacement with @I setting the percentage attributed to the parents.
Operators

SubcubeSwap [@D [@I]] The crossover for swapping subcubes. First parameter is its probability and the second is the size of the swapped subcube.

N-Point [@D [@I]] The n-points crossover. First parameter is its probability and the second number of points.

FlipEdge [@D [@D]] The mutation for flipping activation of random edges. First parameter is its probability and the second is the probability of a particular edge getting flipped.

CleverRepair [@D [@D]] The clever variant of the mutation for repairing non-3-spanners. First parameter is its probability of the mutation and the second is the probability of a particular edge getting repaired.

SimpleRepair [@D [@D]] The simple variant of the mutation for repairing non-3-spanners. First parameter is its probability of the mutation and the second is the probability of a particular edge getting repaired.

Translation [@D [@I]] The mutation for subcube translation. First parameter is its probability and the second is the size of the translated subcube.

Rotation [@D [@I]] The mutation for subcube rotation. First parameter is its probability and the second is the size of the rotated subcube.

Fitness (goal)

Spanner Sets the task to finding a 3-spanner.

Degree Sets the task to reducing the maximal degree.

EdgeDisjoint @I Sets the task to finding at most @I edge-disjoint spanning trees.

One thing that cannot be configured using the configuration file is the calculation of the fitness function. For this, the user has to change directly the code.
### Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PopulationSize</td>
<td>1000</td>
</tr>
<tr>
<td>Dimension</td>
<td>5</td>
</tr>
<tr>
<td>Generations</td>
<td>1000</td>
</tr>
<tr>
<td>EliteSize</td>
<td>0.1</td>
</tr>
<tr>
<td>OutputRate</td>
<td>25</td>
</tr>
<tr>
<td>Parallel</td>
<td></td>
</tr>
</tbody>
</table>

### Initialization

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputFolder</td>
<td>D:\Hypercubes</td>
</tr>
<tr>
<td>Percentage</td>
<td>40</td>
</tr>
<tr>
<td>FileUsage</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### Probability

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation</td>
<td>0.7</td>
</tr>
<tr>
<td>Crossover</td>
<td>0.5</td>
</tr>
<tr>
<td>MutationPerItem</td>
<td>0.01</td>
</tr>
</tbody>
</table>

### Selection

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChildrenOnly</td>
<td></td>
</tr>
<tr>
<td>Tournament</td>
<td></td>
</tr>
</tbody>
</table>

### Operators

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value1</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubcubeSwap</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>CleverRepair</td>
<td>0.02</td>
<td>0.001</td>
</tr>
<tr>
<td>SimpleRepair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translation</td>
<td>0.9</td>
<td>2</td>
</tr>
<tr>
<td>Rotation</td>
<td>0.9</td>
<td>2</td>
</tr>
<tr>
<td>FlipEdge</td>
<td>0.7</td>
<td>0.005</td>
</tr>
</tbody>
</table>

### Fitness (Goal)

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value1</th>
<th>Value2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanner</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Example of a configuration file.

### 6.3 Extending the program

Not everything in the program can be easily modified or extended, but many useful things can and this section will describe the substantial of that.

#### 6.3.1 Settings

The default settings are stored within the properties of the project. One way to modify the setup is to comment or uncomment snippets of code within the `Settings` class. However, we recommend using the configuration file, whose
concept and capabilities might need to be adjusted to the user. For parsing it, we are using these extensions on an array:

```csharp
int ParseIntOrElse(
    this ILList<string> list,
    int index,
    int defaultValue)

double ParseDoubleOrElse(
    this ILList<string> list,
    int index,
    double defaultValue)

TItem GetOrElse(
    this ILList<TItem> list,
    int index,
    TItem defaultValue)
```

The main advantage is that they handle failures both on the number of provided arguments and parsing values, supplying the pre-set default value at both occasions.

### 6.3.2 Operators

All operators must implement the `Operator` interface, which constitutes of two methods:

```csharp
void Operate(Population parents, Population offspring);
void Update();
```

`Operate` takes on input the parents and produces the offspring. `Update` is called for each operator once for each generation. The idea is that the user can use it to modify the settings and parameters of that particular operator, such as change its probabilities or the (fixed) size of objects it operates with.

### 6.3.3 Selector

Similarly to operators, selectors need to implement the `Selector` interface, which has the method `Select(int howMany, Population from, Population to)`. As the signature suggests, it takes `howMany` individuals from `from` and places them into `to`.

### 6.3.4 Fitness

Modifying the code is actually the only way to change the fitness function. To implement a new one, it is necessary to implement the `FitnessFunction` interface, which has the method `double Evaluate(Individual ind, bool count);` that takes an individual and returns its fitness value. The `count` boolean can be ignored; its purpose is to count the spanners for the statistics on output. If needed, it is recommended to set the objective value of the individual within this method.

In many tasks the value of the fitness function differs significantly from the property we are actually interested in. In those tasks, it is useful to set the
**objective function** value to match the desired property. Then the user can easily monitor the development of that property without having to deduce it from the fitness value.

### 6.4 Interpreting Console Output

During the computation the program produces output on the console as seen in Figure 6.2. Each of the vertical lines at the beginning of the lines mean that 20 percent of the generations between two console outputs were calculated. The reason is that for bigger populations and/or higher dimensions the computation is rather slow and the user typically values some feedback to know that the program is still running.

<table>
<thead>
<tr>
<th>Gen</th>
<th>obj</th>
<th>at</th>
<th>fit</th>
<th>3-s</th>
<th>med</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>119</td>
<td>240</td>
<td>76</td>
<td>2,00%</td>
<td>69</td>
</tr>
<tr>
<td>50</td>
<td>111</td>
<td>174</td>
<td>82</td>
<td>2,50%</td>
<td>75</td>
</tr>
<tr>
<td>75</td>
<td>107</td>
<td>235</td>
<td>86</td>
<td>0,90%</td>
<td>79</td>
</tr>
<tr>
<td>100</td>
<td>104</td>
<td>0</td>
<td>88</td>
<td>17,80%</td>
<td>80</td>
</tr>
<tr>
<td>125</td>
<td>104</td>
<td>0</td>
<td>88</td>
<td>32,70%</td>
<td>81</td>
</tr>
<tr>
<td>150</td>
<td>104</td>
<td>0</td>
<td>88</td>
<td>33,80%</td>
<td>81</td>
</tr>
</tbody>
</table>

**Figure 6.2:** Example of console output with **OutputRate** set to 25. The interpretation goes: In generation 25, the best 3-spanner had 119 edges and according to its fitness, which was 76, it was 240th best. In the whole population there were 2% of 3-spanners and the median fitness was 69.

### 6.5 Logging

Logging data is essential for analysis of runtime in GA. The logs, whose output directory can be specified using the **OutputDir** configuration option, contain information about the settings of the GA and about each generation of individuals during runtime. Their formatting is rather simple, but sufficient for the task. Due to parallelism, each run has its own logger that takes care of its output. Since parallel runs, during which only one of the instances is allowed to use console output, are used considerably more often than sequential, logging in *CubeBreeder* is compulsory.
7. GA Experiments and Results

This chapter presents the results obtained through the *CubeBreeder* program, whose features were presented in previous chapters. Section 7.1 discusses the effectiveness of different set-ups and presents empirical data from a few interesting test runs. Section 7.2 contains a comparison of the GA and theoretical results from Chapter 2.

### 7.1 Settings of GA

It is important to note the vast number of combinations of operators we need to evaluate. Moreover, for each combination, different values used for the GA configuration may affect the performance and results significantly. Clearly, it is impossible to gather enough data to support statements about all of them.

Section 7.1 will first discuss a selection of operators and configurations to get a baseline for performance. Next, several tests will be performed and compared to the baseline and with each other. Some tests will be examined in detail, while others, particularly the poorly performing ones, will only be mentioned without supporting the statements with data.

The configuration experiments were conducted with the 3-spanner fitness function, as it worked the best and is also the core part of this thesis. The main tests were run 10 times for each dimension using a different fixed random seed every time. For all the experiments, we used a laptop with Intel i7 CPU and 8GB RAM.

#### 7.1.1 Baseline

A basic set of settings were used to establish the baseline used in the comparison of how different configurations performed. The reasoning behind these specific settings as well as the settings themselves will be explained below. The results of the baseline tests are presented in Table 6.

**Dimension: 4 to 6**

It makes no sense to run the program for dimensions 1 or 2, where the spanner has to contain all the edges. Similarly, for dimension 3, it is easy to find a spanner of desired properties. Therefore, we begin the experiments at dimension 4, when finding a solution is no longer trivial.

On the other hand, dimension 6 is the first number where the tests failed to reach the known upper bound on the number of 3-spanners. Keeping in mind the computational and space complexity of the task, for higher dimensions, only the best performing settings tested were run, see Section 7.2.

**Population Size: 10 · |E(Q_n)|**

Due to the fact that the size of an individual changes significantly with the dimension used, population size must be set accordingly. For example, having a population size of 1000 is too much for dimension 3, but too little for dimension...
6. There needs to be a good ratio set between the running time of the program and the size of the population. With too small a population, the results would be affected by a lack of variety.

Experiments have shown that multiplying the size of the individual by 10 works well enough. Attempts to use the size of an individual turned out to be too little, and multiplying the size of an individual by 50 was too much, although with both these configurations there were some good runs for some dimension.

**Number of Generations:** 1000

Considering the termination criteria, for the sake of comparison it was decided the maximum number of generations be set at 1000. To verify that this number of generations was not too short, an experiment on 2500 generations was conducted to see if the population would change significantly — it did not.

**Initialization Percentage:** 45 %

Since the known upper bound on the number of edges in a 3-spanner is ≈ 45 %, we randomly activate that percentage of edges in the individuals of the initial population. We obviously use random initialization, as the benefits of using a pre-calculated spanner are yet to be explored.

**Elite:** 10 %

Following the experience from testing the program, the elite size was set to 10 %. Running the program without an elite leads to considerable oscillation of values and often the best individuals will appear during the run but not in the last generation(s).

**Selection: Tournament**

Since several fitness functions are being used with very different values and properties during testing, it only makes sense to use the tournament crossover method. This method is less prone to the distribution of fitness within the population. Because there was no intention to set particularly strong selection pressure, the tournament selection was used with two competitors.

During environmental selection the *ChildrenOnly* replacement strategy was used in order to keep the baseline clean of mixing generations.

**Crossover: SubcubeSwap**

Having had designed a crossover specifically for the hypercube, it was obviously used in the baseline configuration. Based on observations during experiment runs, the crossover probability of the subcube swap was set to 0.5 and the fixed subcube size to 2. This size was used as 1 makes little sense and 3 would be too big for dimensions 4 and 5.
Mutation: FlipEdge

Because in the baseline configuration the mutation is the only mean of affecting single edges, it is desired for it to occur frequently enough. Again based on observations during experiment runs, we set the mutation probability to 0.8 and the probability of mutation on an edge to 0.01.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Dim</th>
<th>Obj</th>
<th>Best</th>
<th>#Best</th>
<th>Gen</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic set-up</td>
<td>4</td>
<td>18.9</td>
<td>18</td>
<td>2</td>
<td>90</td>
<td>5 s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>44.2</td>
<td>43</td>
<td>1</td>
<td>125</td>
<td>10 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>100.9</td>
<td>99</td>
<td>2</td>
<td>523</td>
<td>48 s</td>
</tr>
<tr>
<td>Test 1</td>
<td>4</td>
<td>18.5</td>
<td>18</td>
<td>5</td>
<td>117</td>
<td>5 s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>42.8</td>
<td>40</td>
<td>3</td>
<td>183</td>
<td>31 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>99.8</td>
<td>97</td>
<td>1</td>
<td>768</td>
<td>113 s</td>
</tr>
<tr>
<td>Test 2</td>
<td>4</td>
<td>18.2</td>
<td>18</td>
<td>8</td>
<td>213</td>
<td>5 s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>42.7</td>
<td>40</td>
<td>3</td>
<td>255</td>
<td>24 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>99.7</td>
<td>94</td>
<td>1</td>
<td>620</td>
<td>147 s</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>235.3</td>
<td>230</td>
<td>2</td>
<td>910</td>
<td>845 s</td>
</tr>
<tr>
<td>Test 3</td>
<td>4</td>
<td>18.5</td>
<td>18</td>
<td>5</td>
<td>208</td>
<td>5 s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>43.8</td>
<td>43</td>
<td>2</td>
<td>275</td>
<td>27 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>102.1</td>
<td>100</td>
<td>2</td>
<td>843</td>
<td>159 s</td>
</tr>
<tr>
<td>Test 4</td>
<td>4</td>
<td>18.5</td>
<td>18</td>
<td>5</td>
<td>208</td>
<td>4 s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>43.8</td>
<td>40</td>
<td>2</td>
<td>275</td>
<td>25 s</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>101.5</td>
<td>100</td>
<td>2</td>
<td>635</td>
<td>144 s</td>
</tr>
</tbody>
</table>

Table 6: Overview of the test results on 3-spanners with the best test run shown in bold. Dim is the hypercube dimension, Obj the average value of objective function, i.e. the number of edges in the 3-spanner, Best the best value of objective function, #Best the number of occurrences of this best value in the test runs, Gen the average generation of when the best value of each run appeared for the first time and Time to the average runtime in seconds of a single run.

Configuration files, logs and best individuals for the following tests can be found in the Attachment. However, because the CubeBreeder is due to performing experiments a constantly changing tool, running the same configuration will provide slightly different values. The conclusion drawn from the tests remains the same nevertheless.

7.1.2 Test 1

The first test extended the mutation spectrum with translation and rotation of subcubes, which we consider to be the most natural mutations on a hypercube.

Setup The mutations representing automorphisms of a hypercube were added to the baseline configuration. The subcube size within them was set to 2 and the probability to 0.8 as was the FlipEdge mutation.

Outcome We can see in the corresponding row of Table 6 that the performance clearly increased. For dimensions 4 and 5 more individuals with the best
reached objective value were found, and for dimensions 5 and 6 those were even better than in the baseline. Another thing worth noting is the high average generation number for dimension 6 — that is because the individuals were gradually getting better, unlike the baseline, where they reached their best computed value sooner and remained stuck there, although there was clearly room for improvement.

### 7.1.3 Test 2

With the success of Test 1, we tried changing the size of the subcube randomly during runtime.

**Setup** The same as in Test 2, but the subcube size in the crossover and the automorphism mutations can change by one in both directions with probability 0.5.

**Outcome** This is the best performance of all the tests. We attribute it to the fact that with varying subcube size the operations that would take many generations to perform with a small subcube size can be done in a single step.

### 7.1.4 Test 3

The next test, aimed at selection, actually consists of three, however, only results of the last one are presented in Table 6. The first idea was to use Roulette Wheel selection instead of Tournament, but that turned out to be clearly worse. Next, we tested Boltzmann Tournament and discovered that even with different parameters of the temperature function, it behaved almost identically to Test 2. The only remaining selection procedure was the Boltzmann variant of the Roulette Wheel.

**Setup** The same as in Test 2, but with Boltzmann Roulette Wheel instead of Tournament as the selector.

**Outcome** We can see that the performance was not the best, but showed some decent features. For dimension 6, there was again a gradual improvement during the whole run. That is quite a contrast to the standard roulette, which converged to a local optimum fairly quickly.

### 7.1.5 Test 4

So far we used the same crossover, so in this test we changed subcube swapping for an $n$-point crossover. We tried out a few values for $n$ and settled at $n = 2$.

**Setup** Same as Test 2, but with 2-point crossover instead of subcube swap.

**Outcome** Although better than some of the other attempts, it is clear that the specialized crossover works better for the task.
7.1.6 Other tests

Our configuration-testing efforts were not limited to those mentioned above. However, other settings showed behaviour discouraging from conducting the full test.

File Initialization

When initializing with good spanners and using elitism at the same time, clearly, some of the loaded spanners survive through all the generations and represent the best solution. Obviously, if there was a better spanner, it could be created, but that has never been the case for us. On the other hand, we do not consider it to be very likely that there are better spanners than those described in Section 2.3. When not using elitism, the file-initialized spanners die and some generations later the best individual in the population reaches a value worse than that of the spanner in the file, since the original spanner is broken by crossover with randomly initialized ones.

Other Probabilities for FlipEdge

Initially, we believed that using $1/|E(Q_n)|$ is better, since then on average exactly one edge mutates, but experimenting with this probability proved us wrong. Surprisingly, the value of the probability did not seem to depend on the dimension.

Edge Repairing Mutations

Experiments with the repairing mutations showed that they need to be used carefully. Otherwise, they lead to a quick convergence to a local optima. Therefore the probability of the occurrence has to be very low: 0.0001 or even less. On the other hand, if there was a clever mutation to remove unnecessary edges, combining it with the repairing one could possibly bring interesting results. The problem is that knowing what an unnecessary edge looks like would also likely allow us to improve the theoretical results.

Different Values for Fixed-Size Subcube Operators

We observed that the size of the subcube noticeably affects the performance. For the conducted experiments, the difference between dimensions 4 and 6, where subcube of dimension 3 would also be worth considering, is so small that we can assume the effect to be neglectable. However, for higher dimensions it is almost a necessity to use multiple sizes during the GA. If that is not possible for whatever reason, size 2 works for any higher dimension, but requires several generations to perform the same operation than a bigger subcube would.

Replacement Strategies

Experimenting with the replacement strategies showed that their behaviour persists no matter the selection method nor the operators. For most runs, the ChildrenOnly strategy outperformed the other two. In addition, it is the only one not requiring any additional computation (a reference is passed) and thus by far the fastest.
7.2 Graph Theory and GA Comparison

We can conclude the thesis by comparing the theoretical results from Chapters 1 and 2 with CubeBreeder. Throughout the development, we gathered many hypercube spanners. In this section, we do not present results achieved by a particular method, but the best-ever computed values. Some of them might have been a result of a lucky seed or of keeping the program running for many generations, but they were found by CubeBreeder nevertheless. We include the computed 3-spanners from those runs in the attachment. Unfortunately, for some of them we do not possess the configuration we used.

7.2.1 3-spanner Size

We focused mostly on minimizing the size of a 3-spanner; firstly because we consider it the most interesting property and secondly it turned out to be the one most easily solved by GA, probably mostly due to the rather straightforward fitness function that reflected the properties well.

The results are presented in Table 7. Reaching the known bounds is easy for dimensions 3 and 4, and still fairly achievable for dimension 5. Even dimensions 6 and 7 do not require any particular effort — note that the best value for dimensions 6 and 7 was achieved (again) during testing. For higher dimensions the gap to the known theoretical constructions increases. For dimensions 6 and 7 we could easily reach the known upper bound by initializing the population with a 3-spanner of size equal to the upper bound, which we could construct following Lemma 28, but those values are not truly computed and therefore not included in Table 7. Same goes for construction from Lemma 29 and higher dimensions.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Table 4</th>
<th>CubeBreeder</th>
<th>Extra edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>0 %</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>18</td>
<td>0 %</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>40</td>
<td>0 %</td>
</tr>
<tr>
<td>6</td>
<td>88</td>
<td>94</td>
<td>6.8 %</td>
</tr>
<tr>
<td>7</td>
<td>192</td>
<td>230</td>
<td>19.8 %</td>
</tr>
<tr>
<td>8</td>
<td>416</td>
<td>572</td>
<td>37.5 %</td>
</tr>
<tr>
<td>9</td>
<td>896</td>
<td>1322</td>
<td>47.5 %</td>
</tr>
<tr>
<td>10</td>
<td>1920</td>
<td>2956</td>
<td>54.0 %</td>
</tr>
<tr>
<td>11</td>
<td>4096</td>
<td>6439</td>
<td>57.2 %</td>
</tr>
<tr>
<td>12</td>
<td>8448</td>
<td>13686</td>
<td>62.0 %</td>
</tr>
<tr>
<td>13</td>
<td>17408</td>
<td>29526</td>
<td>69.6 %</td>
</tr>
</tbody>
</table>

Table 7: Comparison of the best known upper bound constructions of 3-spanners (Table 4) and evolutionarily computed values.

1We retried the successful testing run of dimension 6 for some more generations, but the value did not improve.
7.2.2 Maximal Degree

While the 3-spanner results are pretty close to the currently known bounds at least for small dimensions, the same cannot be said about minimizing the maximal degree. We believe the reason is that unlike 3-spanners, finding an appropriate fitness function is more difficult for this task. Also, since it is not our main concern, we did not develop specific mutations, such as a “repairing” one that would check if any of the edges of a high-degree vertex can be removed. That would likely again lead to local optimum, but it would certainly improve the results.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Possible Value</th>
<th>CubeBreeder</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.00</td>
<td>2.67</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2.37</td>
<td>3.24</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2.70</td>
<td>3.74</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3.00</td>
<td>4.20</td>
<td>3-4</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>3.27</td>
<td>4.61</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3.53</td>
<td>5.00</td>
<td>4-5</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>3.77</td>
<td>5.36</td>
<td>4-5</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>4.00</td>
<td>5.71</td>
<td>4-5</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 8: Comparison of known values for minimizing maximal degree and evolutionarily computed values

Table 8 shows the known bounds on the minimal maximal degree together with a list of values these bounds allow and the value computed by CubeBreeder.

7.2.3 Edge-Disjoint Spanning Tree

The last property we dealt with was finding multiple edge-disjoint spanning trees. Again, setting up the right fitness was the main issue and the results in Table 9 do suggest we succeeded only partly. The program is able to find two edge-disjoint spanning trees but not more. In the other hand, it does not use all the edges in those two.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Maximum</th>
<th>CubeBreeder</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9: Overview of the maximal number of edge-disjoint spanning trees in $Q_n$ (maximum) and comparison to the computed values.
7.2.4 Summary

It appears that if we use straightforward fitness function, the GA for small dimensions provides interesting results in reasonable time. However, in tasks where the properties of a good solution are difficult to evaluate it fails as expected.

We can conclude that there are certainly problems on hypercubes that can be efficiently studied using genetic algorithms, if they fulfil the condition of having a straightforward fitness function.
## Conclusion

We made an overview of state-of-the-art knowledge in this area and found that the current-most results prove only asymptotic bounds.

We focused briefly on the dominating sets of these asymptotic bounds constructions and adapted and translated into English the relevant part of one significant paper [41], that was available in Russian only. However, we discovered that the results from this paper do not help us with the size of local detour spanners for small dimensions of hypercubes.

Therefore we moved our interest in the direction of small dimension dominating sets. Those are known [48, 49] exactly for \( n \leq 9 \) and within a reasonable interval for \( n = 10 \). Exact values are also known for hypercubes \( Q_n \) where \( n = 2^k \) and \( n = 2^k - 1 \). Using these numbers, we were able to calculate the upper bounds of local detour spanners for dimensions \( n \leq 10 \) as seen in Table 3.

Seeing that the asymptotic constructions from Section 1.3 do not perform well on small dimensions (see Section 2.4), we looked into constructions designed specifically for small dimensions. We found two in [51] and described them in Section 2.3, offering a comparison with the asymptotic construction in Table 4.

Local detour subgraphs were not our sole interest. We briefly reviewed the maximal degree of detour spanners and edge-disjoint spanners for the purpose of the genetic algorithms. Based on the results found in regard to the delays of edge-disjoint spanners in hypercubes from [22], we deduced the consequences for the second part of the thesis.

The second part of this thesis explores the possibilities of using genetic algorithms for the above mentioned hypercube problems. Analysis is performed to the conclusion that computational challenges are the limit of GA efforts. Therefore the main aim is to generate spanners with desired properties — either 3-spanner property, small maximal degree or multiple edge-disjoint spanners — for small dimensions only.

A representation for hypercube subgraphs in a GA is presented and fitting selectors as well as and operators are designed. The operators cover all the logical operations that can be performed on a hypercube: changing a subcube and, within some generations, doing any possible automorphism on a subcube.

Having set the theoretical design foundations, the GA is implemented and its performance is evaluated. The results show that it is a viable approach, but highly limited by computational complexity and runtime for higher dimensions, and the difficulty of the problem itself. The GA gives a better outcome if it fulfils the condition of having a straightforward fitness function. The program can easily be adapted to using GA in other hypercube problems.

## Future Work

There are several possible extensions and open problems, both theoretical and computational, that can be explored in future work. Although the program can be adapted to potentially solving many problems related to hypercubes, we restrict the recommendations for future work to topics already examined in this thesis.

Of the theoretical ones, although it does not seem to be the case very likely,
we wonder if, similarly to the constructions in Section 2.3, which use hypercubes of size $n = 3$ and $n = 7$, there is also a good construction using hypercubes with $n = 15$, since that is the next dimension with a perfect dominating set.

Concerning the combination of hypercubes and genetic algorithms, we see potential in combining multiple properties for small dimension hypercubes. Another interesting topic that could be explored are the edge-disjoint spanners. Lemma 19 gives us a set of edge-disjoint spanners, where only one spanner has some guaranteed delay. If longer delay checks are implemented into the CubeBreeder, it could be used to find more edge-disjoint spanners with a guaranteed delay.
Bibliography


### List of Figures

1. Examples of hypercubes $Q_n$ for $n = 1, 2, 3, 4, 5$ generated using [1]. 4
2. Illustration of hypercube graphs. Note the labelling of the vertices with fixed-length binary string, which will be used throughout the thesis. Image generated using [1]. ................. 4
3. A schema showing the general concept of a genetic algorithm. .................. 7

1.1 Hypercube $Q_4$ with a $Q_3$ subcube (yellow). ....................... 10
1.2 (2,1)-detour example. ........................................... 14
1.3 An example of two edge-disjoint spanning trees in $Q_4$. The edges of the first spanning tree are in green and those of the second one in yellow. Note that such a construction does not preserve the distance well. The two vertices marked by blue colour are adjacent in $Q_4$, but have distance 5 in the green spanner. ............... 16

2.1 A 3-spanner of $Q_7$. The edges between the copies of $Q_5$ are all in the spanner, but are omitted in the picture, except the edges from vertices $u$ and $v$. The grey areas represent all edges between two consecutive levels of $Q_5$. Note that this is an example of a 3-spanner that is not an additive 4-spanner. The picture is courtesy of [51] ......................... 23

3.1 A simple genetic algorithm idea in a pseudo-code. ..................... 28
3.2 The fitness landscape of a multi-objective fitness function. .......... 28
3.3 Example of binary representation .................................. 29
3.4 Example of integer representation .................................. 29
3.5 Example of permutation representation ............................... 29
3.6 Example of floating point representation ............................. 29
3.7 Roulette Wheel selection ............................................. 31
3.8 Stochastic universal sampling selection ............................... 31
3.9 Tournament selection with four competitors .......................... 32
3.10 $n$-point crossover examples ...................................... 34
3.11 Examples of other crossover methods ................................. 34
3.12 Example of a crossover with three parents .......................... 35
3.13 Examples of bit-flipping mutations .................................. 35
3.14 Example of a 2-opt mutation. The crossed out edges are changed for the red ones. ............................................. 36
3.15 The effect of crossover on the position of an individual in fitness landscape. ......................................................... 37

4.1 Subcube Swap Example .............................................. 42

5.1 Summary of the Edge class .......................................... 48
5.2 Summary of the GraphInfo class .................................... 49
5.3 Summary of the Individual class .................................... 49
5.4 Illustration of the layers of the Subcube Swap Crossover with fixed bits at indices 2 and 3 with fixed value 0. The “outer” layer is in blue, the “inner” layer in green and the connecting edges in yellow. 50
5.5 The core of the algorithm intelligently repairing an invalid 3-spanner 51
5.6 The algorithm to check if a subgraph is a 3-spanner .......... 52
5.7 The algorithm to count components in the subgraph .......... 53
5.8 DFS Colouring algorithm ........................................ 53
5.9 Input / Output formatting of the subgraph of $Q_3$. Note the order of appearance of the edges ............. 54

6.1 Example of a configuration file ................................. 60
6.2 Example of console output with OutputRate set to 25. The interpretation goes: In generation 25, the best 3-spanner had 119 edges and according to its fitness, which was 76, it was 240th best. In the whole population there were 2% of 3-spanners and the median fitness was 69 ............................. 62
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sizes of dominating sets in hypercubes of small dimension.</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Values of $\gamma(Q_n)$ for example dimensions $n = 2^k - 1$ and $n = 2^k$.</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Values for the upper bound on the number of edges in the minimal local detour spanners in hypercubes of small dimension.</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Comparison of values for upper bound on local detour spanners in hypercubes of small dimension.</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Comparison of values for upper bound on local detour spanners in hypercubes of some higher dimensions.</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Overview of the test results on 3-spanners with the best test run shown in bold. Dim is the hypercube dimension, Obj the average value of objective function, i.e. the number of edges in the 3-spanner, Best the best value of objective function, #Best the number of occurrences of this best value in the test runs, Gen the average generation of when the best value of each run appeared for the first time and Time to the average runtime in seconds of a single run.</td>
<td>65</td>
</tr>
<tr>
<td>7</td>
<td>Comparison of the best known upper bound constructions of 3-spanners (Table 4) and evolutionarily computed values.</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>Comparison of known values for minimizing maximal degree and evolutionarily computed values.</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>Overview of the maximal number of edge-disjoint spanning trees in $Q_n$ (maximum) and comparison to the computed values.</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Abbreviations

**EDS**  Edge-disjoint spanner — for definition see Section 1.6.

**EA**  Evolutionary algorithm — for further explanation see Chapter 3.

**GA**  Genetic algorithm — for further explanation see Chapter 3.

**SGA**  Simple genetic algorithm — for further explanation see Section 3.4.
Attachment

We attach a CD with the following contents:

**CubeBreeder** The source code of the program.

**BestRuns** A folder with some of the best achieved results.

**Tests** Configuration files, logs and outputs of the tests described in Section 7.1.

**spanner-generator.py** A Python script for generating small dimension 3-spanners as described in Section 2.3.

**config.txt** A configuration file.