Università degli Studi di Napoli Federico II

Solving Parity Games in Practice

A Walkthrough Across New Technologies

Vincenzo Prignano

Scuola Politecnica e delle Scienze di Base
Area Didattica di Scienze Matematiche Fisiche e Naturali
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Tesi Triennale Sperimentale in Informatica

Relatore: Prof. Aniello Murano

A.A. 2013/2014

To my Family.

0.1 Abstract

Parity Games are simple two player games based on perfect information, played on directed graphs, where every node is priority-labeled. The study of this kind of games is an interesting open problem in computer science. Solving parity games is a known to be polynomial time, equivalent to model checking for the modal µ-calculus. The problem also belongs to the complexity class $NP \cap co - NP$. Through these years many algorithms have been proposed to solve this kind of games, but the theoretical knowledge is not yet matched to the practical efforts of this area. The purpose of this work is a deep investigation among modern programming languages such as Scala, Java 8, C++11 and Go using a slightly improved version of the recursive algorithm [10]. The achieved result is a significant improvement of performance (that can be estimated around two order of magnitude in running time), the use of modern languages, multiple applications able to solve randomly generated and special games and a base ready to be extended for future works, improvements and new algorithms.

Contents

1 Introduction

Parity games [11, 41] are abstract infinite-duration are two-person infinite-durable (player 0 and 1) graph based games, that represents a powerful mathematical framework to address fundamental questions in computer science and mathematics. Infinite games are proved to be a useful method to describe the runs of automata on infinite trees. A game consists in two sets of nodes that are owned by a player and such games are determined: either player 0 or player 1 wins a chosen node in a game, the determinacy of such games follows from [27] Borel games (a class of games that includes parity games) are determined. In this research thesis we deal only with parity games with fixed number of nodes.

In the basic settings, parity games are two-player turn-based games, played on directed graphs, whose nodes are labeled with *priorities* (i.e., natural numbers). The players, named *player* 0 and *player* 1, move in turn a token along the edges. A play may result in an infinite path. Player 0 wins the play if the greatest priority is even, otherwise player 1 wins the game.

Condition	Complexity
Recursive [41]	$O(e \cdot n^d)$
Small Progress Measures [20]	$O(d \cdot e \cdot (\frac{n}{d})^{\frac{d}{2}})$
Strategy Improvement [40]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition [21]	$O(n^{\sqrt{n}})$
Big Step [34]	$O(e \cdot n^{\frac{1}{3}d})$

Table 1.1: Parity algorithms along with their computational complexities.

Given a model a system and a defined a specification, the *model checking problem* is the problem of deciding if the specifications are satisfied by the system. Defining specifications include safety and liveness properties, the former outlines a state where nothing bad happened while the latter outlines a state where good things may eventually happen.

The *modal μ-calculus*, a fixed point logic of programs has been introduced by Kozen in [22], is a way to express such properties. The close relationship between parity games and *μ-calculus* is remarkable to understand the gaining momentum of parity games. There are linear reductions between *μ-calculus* model checking problem and problem to solve parity games. In other words, from a complexity point of view, solving a parity game is equivalent, under linear-time reductions to the introduced model checking problem, hence any tool for solving parity games is also a model checker for the *μ-calculus*. In formal system design and verification [9, 23], parity games arise as a natural evaluation machinery to automatically and exhaustively check for reliability of distributed and reactive systems [3, 5, 24].

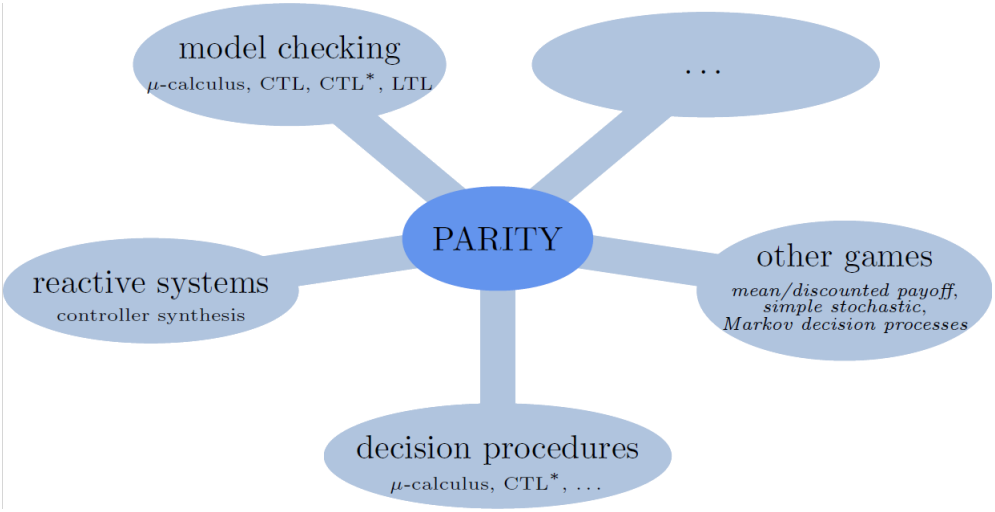

Figure 1.0.1: Applications of Parity Games

Another reason of gaining interest may rely on their complexity status. The problem of solving a parity game is part of the few natural problems included in the $UPTime \cap CoUPTime$ class [19].

Deciding whether a polynomial time solution exists or not is a long standing open question, in fact through the years several algorithms, to solve parity games, have been proposed. Through these years a variety of algorithms for solving parity games has been presented: Zielonka [41] yields a recursive algorithm, Jurdziński's small progress measures algorithm [20], the strategy improvement algorithm by Jens Vöge and Marcin Jurdziński [40], big-step by Schewe [34], etc. It's worth of note that this variety is completely owed entirely to the actual theoretical challenge rather than practical reasons.

In addition to the multitude of invented algorithms, the literature contains few improvements on how to tune a solver, for example Jurdziński suggests to perform SCCs decomposition of a graph [20]. SCC stands for *strongly connected component*, an SCC is a non empty set of nodes in a graph with the property that every node in the component can reach each other. It is reasonable to assume that this kind of decomposition helps to speed up the solvers. Let's say that G is a graph and we call $f(G)$ the time that takes to complete the solving process and also that the graph can be decomposed in n SCCs (C_0, ..., C_n), knowing that a parity game can be solved *SCC-wise* means that it will take $f(C_0) + ... + f(C_n) + O(|G|)$. Other optimizations are related to detection of special cases such as *self-cycle games*, *one-parity* game, *one-player* game, and priority compression and propagation.

The step of priority compression attempts to reduce the number of priorities in a parity game, while the priority propagation aimed to increase priority but to reduce the range of priorities in a game and therefore compress the overall priorities. In Table 1.1, are reported some solvers and their known computational complexities, where parameters n, e, and d denote the number of nodes, edges, and priorities in the game, respectively (for more details, see [12, 13]).

All above mentioned algorithms have been implemented in *PGSolver*, a collection of tools to solve, benchmark and generate parity games. The tool was written

in *OCaml* by Oliver Friedman and Martin Lange [12, 13]. This tool is from some years now, the *de-facto* tool for solving parity games. The tool itself allowed to explore the previously hidden area of practical process. For example, contrary to common believe, an increasing large number of priorities does not necessarily impose a great difficulty in practice, this observation was also confirmed in [10].

The recursive algorithm, from Zielonka's work, was declared the best performing one when compared to the other algorithms if no optimizations or preprocessing steps were applied. Also, SCC decomposition was proven to be highly profitable alongside the elimination of self-cycles. It's important to note that not every optimization can speed up every algorithm, for example the recursive algorithm can achieve best result from the elimination of self-cycles and priority compression, this is highly reasonable due to the fact that without self-cycle elimination it would require more recursive calls. The tool also includes a reduction of parity games to SAT formulas due to Friedmann. Moreover, PGSolver offers the possibility to integrate user-written solvers (written in OCaml) into PGSolver.

Another effort in speeding up the solving process was previously attempted by Philipp Hoffmann and Michael Luttenberger in [16], the idea behind was to use a *GPU*, graphics processing units, to solve parity games. A GPU can excel at problems that can be easily split into a large number of parallels tasks. A modern GPU consists of several multi-processors which act independently of each other. In their paper they present a GPU-enabled implementation for solving parity games, in details they implemented small progress measure (SPM), the recursive algorithm and a variant of strategy iteration. The remarkable effort of using a GPU based implementation was made possible choosing *Nvidia* as GPU vendor and using the *CUDA* framework to write the algorithms. For storing the graph and the node information they used multiple arrays, and before starting the actual solving process, their implementation applies multiple preprocessing steps.

Despite the gaining interest in finding efficient algorithms for solving parity games, less emphasis has been put on the *choice* of the programming language itself. The scientific community relied on OCaml as the best performing language to be used in this settings. In [10] we introduced an Improved Recursive Algorithm

and presented a tool written in Scala to solve parity games, that was capable of gaining up to two orders of magnitude in running time compared to PGSolver. Classical Zielonka algorithm requires to decompose the graph game into multiple smaller arenas, which is done by computing, in every recursive call, the *difference* between the current graph and a given set of nodes.

Remarkably, the improved version guarantees that the original arena remains immutable tracking the removed nodes in every subsequent call and checking, in constant time, whether a node needs to be excluded or not. Casting this idea in the above automata reasoning, it is like enriching the state space with two flags ($removed$, $\neg removed$), instead of performing a complementation.

The study presented in this thesis intends to study multiple programming languages, providing a deep comparison using benchmarking tools and concludes presenting a new tool framework. Because the problem of solving this games has been of notable interest to the scientific community, many interesting benchmarks and cases has been studied obtaining as a result relevant conclusions.

The main reason behind this breakdown derives also from a desire to find a programming language that is easy to learn and use, provides robustness, performance and concurrency features. The aim is to achieve even faster solving process, as a first step of contribution to the many improvements available and the one derived from this work.

2 Preliminaries

A *parity game* can be defined as a tuple $G = (V, V_0, V_1, E, \Omega)$ where:

- (V, E) forms a directed graph whose set of nodes is partitioned into $V = V_0 \cup V_1$,
- V_0 and V_1 are two non empty sets of nodes, where $V_0 \cap V_1 = \emptyset$,
- $\Omega : V \to N$ is the *priority function* that assigns to each node a natural number called the *priority* of the node.

We assume E to be *total*, i.e. for every node $v \in V$, there is a node $w \in V$ such that $(v, w) \in E$. In the following we also write vEw in place of $(v, w) \in E$ and use $vE := \{w \mid vEw\}$.

Parity games are played between two players called *player 0* and *player 1*. Starting in a node $v \in V$, both players construct an infinite path (the *play*) through the graph as follows. If the construction reaches, at a certain point, a finite sequence $v_0...v_n$ and $v_n \in V$ then player i selects a node $w \in v_nE$ and the play continues with the sequence $v_0...v_nw$.

Every play has a unique winner, defined by the priority that occurs infinitely often. Precisely, the *winner* of the play $v_0v_1v_2...$ is player i if and only if $max\{\, p \mid \forall j \,.\, \exists k \geq j \,:\, \Omega(v_k) = p \,\}\ mod\,(2) = i$.

Strategy A *strategy* for player i is a partial function $\sigma : V^*V \to V$, such that, for all sequences $v_0...v_n$ with $v_{j+1} \in v_jE$, with $j = 0, ..., n-1$, and $v_n \in V_i$ we have that $\sigma(v_0...v_n) \in v_nE$. A play $v_0v_1...$ *conforms* to a strategy σ for player i if, for all j we have that, if $v_j \in V_i$ then $v_{j+1} = \sigma(v_0...v_j)$.

A strategy σ for player i (σ_i) is a winning strategy in node v if player i wins every play starting in v that conforms to the strategy σ. In that case, we say that player i *wins* the game G starting in v. A strategy σ for player i is called *memoryless* if, for all $v_0...v_n \in V^*V_i$ and for $w_0...w_m \in V^*V_i$, we have that if $v_n = w_m$ then $\sigma(v_0...v_n) = \sigma(w_0...w_m)$. That is, the value of the strategy on a path only depends on the last node on that path. Starting from G we construct two sets $W_0, W_1 \subseteq V$ such that W_i is the set of all nodes v such that player i wins the game G starting in v. Parity games enjoy *determinacy* meaning that for every node v either $v \in W_0$ or $v \in W_1$ [11].

Solving a given parity game means to compute the sets W_0 and W_1, as well as the corresponding *memoryless* winning strategies, σ_0 for player *0* and σ_1 for player *1,* on their respective winning regions. The construction procedure of winning regions makes use of the notion of *attractor*. Formally, let $U \subseteq V$ and $i \in \{0, 1\}$.

Attractor The i-attractor of U is the least set W s.t. $U \subseteq W$ and whenever $v \in V_i$ and $vE \cap W \neq \emptyset$, or $v \in V_{1-i}$ and $vE \subseteq W$ then $v \in W$. Hence, the i-attractor of U contains all nodes from which player i can move "towards" U and player $1 - i$ *must* move "towards" U. The i-attractor of U is denoted by $Attr_i(G, U)$. Let A be an arbitrary attractor set.

The game $G \setminus A$ is the game restricted to the nodes $V \setminus A$, i.e. $G \setminus A = (V \setminus A, V_0 \setminus A, V_1 \setminus A, E \setminus (A \times V \cup V \times A), \Omega_{V \setminus A})$. It is worth observing that the totality of $G \setminus A$ is ensured from A being an attractor.

Formally, for all $k \in \mathbb{N}$, the i-attractor is defined as follows:

$$Attr_i(U) = \bigcup_{k=0}^{\infty} Attr_i(U)^k$$

$$\begin{aligned} Attr_i(U)^{k+1} = {} & Attr_i(U)^k \\ & \cup \{v \in V_i \mid \exists\, (v, w) \in E : w \in Attr_i(U)^k\} \\ & \cup \{v \in V_{1-i} \mid \forall\, (v, w) \in E : w \in Attr_i(U)^k\} \end{aligned}$$

$$Attr_i(U) = \bigcup_{k=0}^{\infty} Attr_i(U)^k$$

2.0.1 The Zielonka Recursive Algorithm

In this section, we describe the Zielonka Recursive Algorithm using the basic concepts introduced in the previous chapter and some observations regarding its implementation in PGSolver.

The algorithm to solve parity games introduced by Zielonka comes from McNaughton's work [28]. The Zielonka Recursive Algorithm, listed in Algorithm 2.1, makes use of a divide and conquer technique. It constructs the winning sets for both players using sub-games removing the nodes with the highest priority from the game and the ones attracted. The algorithm $win(G)$ takes a graph G as input and, after a number of recursive calls over ad hoc built sub-games, returns the winning sets (W_0, W_1) for player 0 and player 1, respectively. The running time complexity of the Zielonka Recursive Algorithm is reported in Table.

Algorithm 2.1 Zielonka Recursive Algorithm

```
function win(G):
    if V == ∅:
        (W0, W1) = (∅, ∅)
    else:
        d = maximal priority in G
        U = { v ∈ V | priority(v) = d }
        p = d % 2
        j = 1 − p
        A = Attr_p(U)
        (W0', W1') = win(G \ A)
        if W_j' == ∅:
            W_p = W_p' ∪ A
            W_j = ∅
        else:
            B = Attr_j(W_1^j)
            (W0', W1') = win(G \ B)
            W_p = W_p'
            W_j = W_j' ∪ B
    return (W0, W1)
```

2.0.2 An example of Parity Game

In this section we provide an example of the solving process of a parity game. In figure 2.0.1 we have a simple graph with eighth nodes, every node is labelled with a priority and the shape of a node distinguish the player. Starting from the upper left node, labelled with 4, we are able to solve this game in 6 steps, where the final one clearly splits the game in two sets, one set of winning nodes for each player.

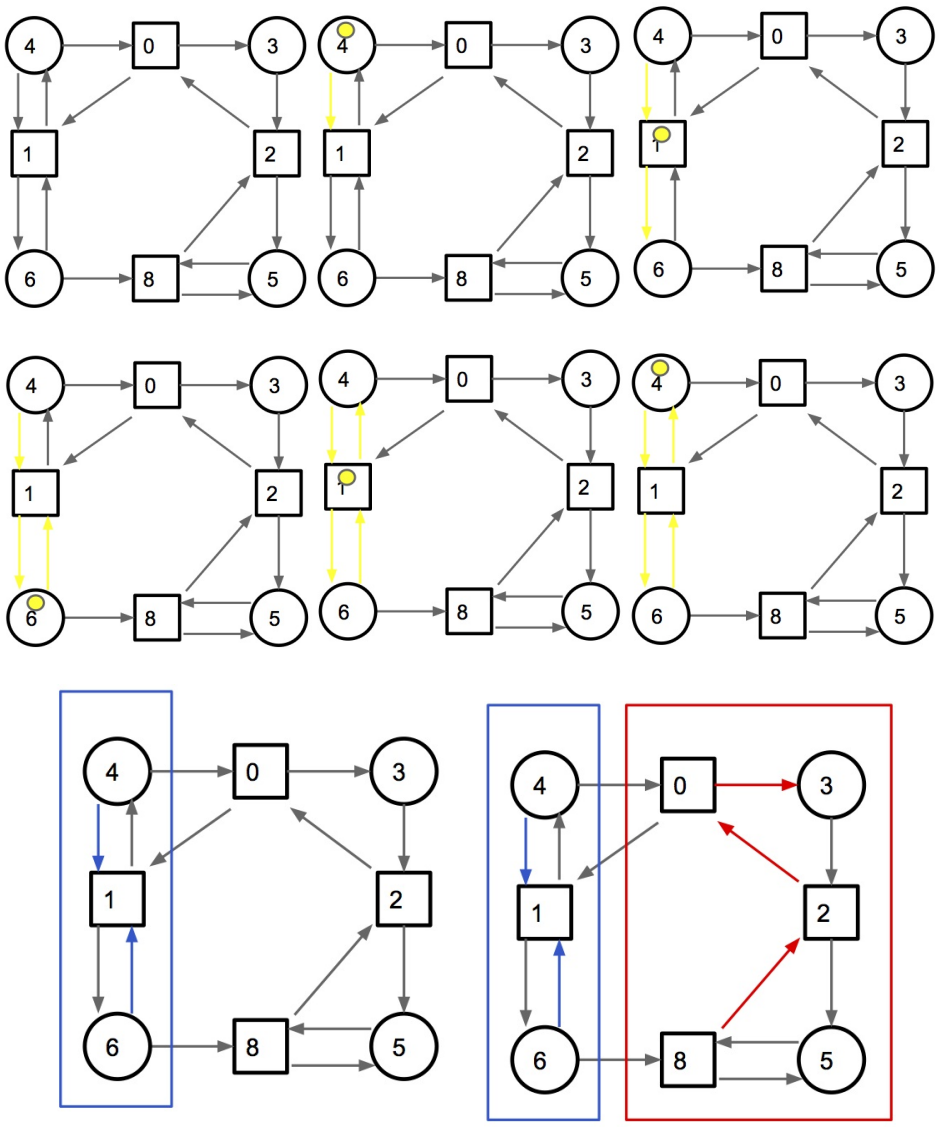

Figure 2.0.1: An example solving a game

3 Analysis

3.1 PGSolver

PGSolver is a tool developed by Oliver Friedman and Martin Lange [12, 13]. This tool is from some years now, the *de-facto* tool for solving parity games. It contains implementations in *OCaml* of about 10 algorithms, including the one implemented in this study: Zielonka's Recursive Algorithm [41]. For benchmarking purposes, PGSolver comes with tools to generate different kind of games, from random to special cases, and lets customize the kind of the game wanted passing as command line flags the number of nodes, available priorities, minimum and number of edges. In [10] we used PGSolver as a comparison to generate and then benchmark multiple games. PGSolver offers many optimizations techniques found in literature for example Jurdziński suggests to perform SCCs decomposition of a graph [20]. Other optimizations are related to detection of special cases such as *self-cycle games*, *one-parity* game, *one-player* game, and priority compression and propagation. The step of priority compression attempts to reduce the number of priorities in a parity game, while the priority propagation aimed to increase priority but to reduce the range of priorities in a game and therefore compress the overall priorities.

The tool itself allowed to explore the previously hidden area of practical process. For example, contrary to common believe, an increasing large number of priorities does not necessarily impose a great difficulty in practice, this observation was also confirmed in [10]. The recursive algorithm, from Zielonka's work, was declared the best performing one when compared to the other algorithms if no optimizations or preprocessing steps were applied. Also, SCC decomposition was proven to

be highly profitable alongside the elimination of self-cycles. It's important to note that not every optimization can speed up every algorithm, for example the recursive algorithm can achieve best result from the elimination of self-cycles and priority compression, this is highly reasonable due to the fact that without self-cycle elimination it would require more recursive calls.

This work is a result of a deep analysis of PGSolver's capabilities in solving parity game in an efficient and performant manner. In more details, even using the Zielonka's Recursive Algorithm, with SCC decompositions enabled PGSolver would require minutes to decide games with few thousands of nodes, especially on dense graphs. Our investigation starts with the way Zielonka's Recursive has been implemented: the graph data structure is represented as a fixed length *Array* of *tuples*, where every tuple contains information about a node, such as the player, priority and adjacency list. Before every recursive call is performed, the implementation performs the difference between the actual graph and the attractor set, outputting a new graph as well as building the transposed graph. In addition the attractor function implemented in PGSolver uses a *TreeSet* as data structure guaranteeing only logarithmic search, inserts and removals. One may say that the added complexity for making a new graph or building the transposed is still linear time on the actual graph, but it is worth noting that general-purpose memory allocators are very expensive as the per-operation cost floats around one hundred processor cycles [14]. Through these years many efforts have been made to improve memory allocation writing custom allocators from scratch, a process known to be difficult and error prone [7, 8][7, 8].

3.2 Improving Recursive

In our previous work [10] we introduced some improvements making a slightly modification to the recursive algorithm. Figure 3.2.1 shows the main problems encountered when working with the recursive algorithm in PGSolver.

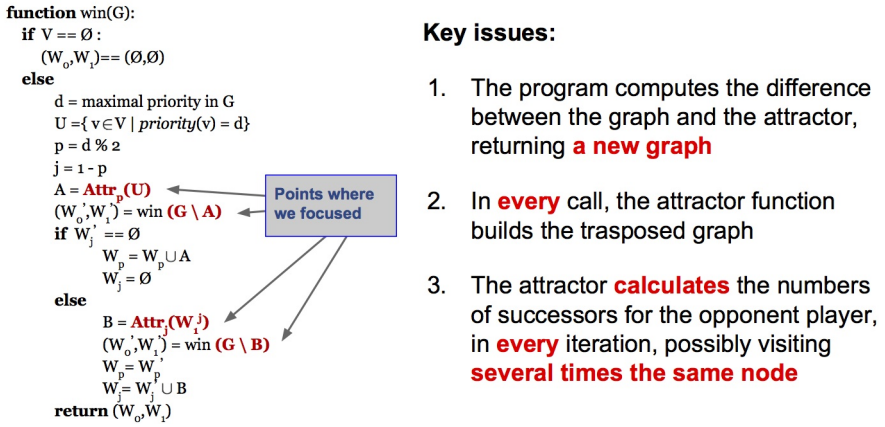

Figure 3.2.1: Improving Recursive Key Points

With this research directions in mind and the aim of performance optimizations, a slightly improved version of the recursive algorithm has been listed as Algorithm 3.1 and its attractor function as Algorithm 3.2.

Algorithm 3.1 Improved Recursive Algorithm Pseudocode

```
function win (G):
    T = G.transpose()
    Removed = {}
    return winImproved(G, T, Removed)

function winImproved(G, T, Removed):
    if |V| == |Removed|:
        return (∅, ∅)
    W = (∅, ∅)
    d = maximal priority in G
    U = { v ∈ V | priority(v) = d }
    p = d % 2
    j = 1 − p
    W' = (∅, ∅)
    A = Attr(G, T, Removed, U, p)
    (W'0, W'1) = winImproved(G, T, Removed ∪ A)
    if W'j == ∅:
        Wp, Wj = W'p ∪ A, ∅
    else:
        B = Attr(G, T, Removed, W'j, j)
        (W'0, W'1) = winImproved(G, T, Removed ∪ B)
        Wp, Wj = W'p, W'j ∪ B
    return (W0, W1)
```

Removing a node from G and building the transposed graph takes time $\Theta(|V|+|E|)$. Thus dealing with dense graph such operation takes $\Theta(|V|^2)$. In order to reduce the running time complexity caused by these graph operations, we a requirement for immutability of the graph G ensuring that every recursive call uses the graph without applying any modification to the state of the graph. Therefore, to construct the sub-games, in the recursive calls, we keep track of each node that is going to be removed from the graph, adding all of them to a set called Removed.

Algorithm 3.2 Improved Attractor Algorithm Pseudocode

```
function Attr(G, T, Removed, A, i):
    tmpMap = []
    for x = 0 to |V|:
        if x ∈ A tmpMap = 0
        else tmpMap = −1
    index = 0
    while index < |A|:
        for v0 ∈ adj(T, A[index]):
            if v0 ∉ Removed:
                if tmpMap[v0] == −1:
                    if player(v0) == i:
                        A = A ∪ v0
                        tmpMap[v0] = 0
                    else:
                        adj_counter = −1
                        for x ∈ adj(G, v0):
                            if (x ∉ Removed):
                                adj_counter += 1
                        tmpMap[v0] = adj_counter
                        if adj_counter == 0:
                            A = A ∪ v0
                else if (player(v0) == j
                            and tmpMap[v0] > 0):
                    tmpMap[v0] −= 1
                    if tmpMap[v0] == 0:
                        A = A ∪ v0
    return A
```

The improved algorithm is capable of checking if a given node is excluded or not in constant time as well as it completely removes the need for a new graph in every recursive call. It turns out to be very successful in practice as proved

in [10] using Scala with respect to PGSolver and also in the following sections, where we provide multiple implementations of the improved algorithm in modern programming languages such as Java 8, C++ 11 and Go.

3.3 Solvers Design and Outline

The frameworks introduced in the following sections, in multiple programming languages, follow a simple design implementation showed, as an UML Class Diagram, in Figure 3.3.1. All implementations make use of the strategy design pattern, guaranteeing an high level of openness and extensibility.

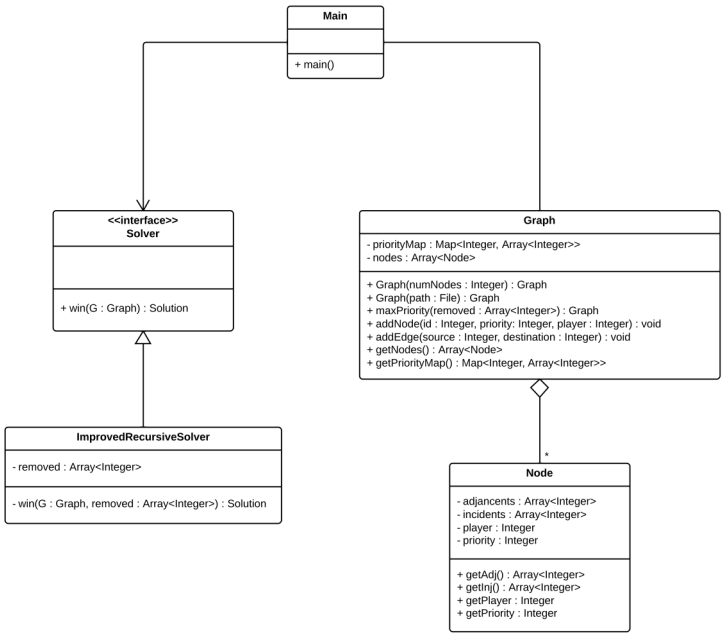

Figure 3.3.1: UML Diagram

In the following sections will be presented multiple implementations in Scala, Java 8, C++11 and Go, every section will introduce the language used, making an overall explanation of the key aspects, the implementation details will then be followed by some benchmarks. Our experiments have been run on multiple instances of random parity games. It's worth to note that these implementations don't apply any preprocessing steps to the arena before solving.

4 Scala

4.1 Introduction

Scala [29, 30] is the programming language designed by Martin Odersky, the co-designer of Java Generics and main author of *javac* compiler. Scala defines itself as a *scalable* language, statically typed, a fusion of an object-oriented language and a functional one. It runs on the *Java Virtual Machine* (JVM) and supports every existing Java library. Scala is a purely object-oriented language in which, like Java and Smalltalk, every value is an object and every operation is a method call. In addition Scala is a functional language where every function is a first class object, also is equipped with efficient immutable and mutable data structures, with a strong selling point given by Java interoperability. However, it is not a purely functional language as objects may change their states and functions may have side effects. The functional aspects are perfectly integrated with the object-oriented features. The combination of both styles makes possible to express new kinds of patterns and abstractions. All these features make Scala programming language as a clever choice to solve these tasks, in a strict comparison with other programming languages available such as C, C++ or Java.

The Scala compiler *scalac* compiles a Scala program into Java class files. The compiler is organized in a sequence of successive steps. The first one is called *the front-end step* and performs an analysis of the input file, makes sure that is a valid Scala program and produces an attributed abstract syntax tree (AST); the *back-end step* simplifies the AST and proceeds to the generation phase where it produces the actual class files, which constitute the final output. Targeting the JVM, the Scala Compiler checks that the produced code is type-correct in order

to be accepted by the JVM bytecode verifier.

In [17], published by *Google*, Scala even being an high level language, performs just 2.5x slower than C++ machine optimized code. In particular it has been proved to be even faster than Java. As the paper notes: "*While the benchmark itself is simple and compact, it employs many language features, in particular high level data structures, a few algorithms, iterations over collection types, some object oriented features and interesting memory allocation patterns*".

4.2 Implementation

Algorithm 4.1 Scala Graph Class

```
1  class Graph extends Cloneable {
2    val priorityMap = new TIntObjectHashMap[ArrayBuffer[Int]]()
3    var nodes = Array[Graph.Node]()
4    var exclude : Array[Boolean] = null
5
6    def this(numNodes: Int) {
7      this()
8      nodes = Array.fill[Graph.Node](numNodes)(new Graph.Node)
9      exclude = Array.fill[Boolean](numNodes)(false)
10   }
11
12   def addNode(node: Int, parity: Int, player: Int) { ... }
13
14   def addEdge(origin: Int, destination: Int) { ... }
15
16   def --(set: ArrayBuffer[Int]) : Graph = {
17     val G : Graph = this.clone().asInstanceOf[Graph]
18     G.exclude = this.exclude.clone()
19     set.foreach(x => {
20       G.exclude(x) = true
21     })
22     G
23   }
24 }
```

Algorithm 4.2 Scala Graph Companion Object

```
1  object Graph {
2    def apply() : Graph = {
3      new Graph()
4    }
5
6    def apply(nodes : Int) : Graph = {
7      new Graph(nodes)
8    }
9
10   class Node {
11     var player = -1
12     var priority = -1
13     val adj = new TIntArrayList()
14     val inj = new TIntArrayList()
15
16     lazy val ~> = adj.toArray
17     lazy val <~ = inj.toArray
18   }
19 }
```

Algorithm 4.3 Scala Solver Interface

```
1  trait Solver {
2    def win(G: Graph) : (ArrayBuffer[Int], ArrayBuffer[Int])
3  }
4
5  object ConcreteSolver {
6    def solve(G : Graph, solver : Solver) :
7       (ArrayBuffer[Int], ArrayBuffer[Int]) = {
8      solver.win(G)
9    }
10 }
```

Algorithm 4.4 Scala Attractor Function

```
1  private def Attr(G: Graph, A: ArrayBuffer[Int], i: Int)
2       : ArrayBuffer[Int] = {
3      val tmpMap = Array.fill[Int](G.nodes.size)(-1)
4      var index = 0
5      A.foreach(tmpMap(_) = 0)
6      while (index < A.size) {
7        G.nodes(A(index)).<~.foreach(v0 => {
8          if (!G.exclude(v0)) {
9            val flag = G.nodes(v0).player == i
10           if (tmpMap(v0) == -1) {
11             if (flag) {
12               A += v0
13               tmpMap(v0) = 0
14             } else {
15               val tmp = G.nodes(v0).~>.count(x => !G.exclude(x)) - 1
16               tmpMap(v0) = tmp
17               if (tmp == 0) A += v0
18             }
19           } else if (!flag && tmpMap(v0) > 0){
20             tmpMap(v0) -= 1
21             if (tmpMap(v0) == 0) A += v0
22           }
23         }
24       })
25       index += 1
26     }
27     A
28   }
```

Algorithm 4.5 Scala Win Function

```
override def win(G: Graph) : (ArrayBuffer[Int], ArrayBuffer[Int]) = {
    val W = Array(ArrayBuffer.empty[Int], ArrayBuffer.empty[Int])
    val d = G.d()
    if (d > -1) {
      val U = G.priorityMap.get(d).filter(p => !G.exclude(p))
      val p = d % 2
      val j = 1 - p
      val W1 = Array(ArrayBuffer.empty[Int], ArrayBuffer.empty[Int])
      val A = Attr(G, U, p)
      val res = win(G -- A)
      W1(0) = res._1
      W1(1) = res._2
      W1(j).size match {
        case 0 =>
          W(p) = W1(p) ++= A
          W(j) = ArrayBuffer.empty[Int]
        case _ =>
          val B = Attr(G, W1(j), j)
          val res2 = win(G -- B)
          W1(0) = res2._1
          W1(1) = res2._2
          W(p) = W1(p)
          W(j) = W1(j) ++= B
      }
    }
    (W(0), W(1))
  }
```

4.3 Benchmarks

All tests in this section have been run on an Intel(R) i7(R) CPU @ 3.9GHz, with 32GB of Ram DDR3 1600Mhz. Precisely, we have used 20 random arenas generated through PGSolver of each of the following types, given $N = i \times 1000$ with i integer and $1 \leq i \leq 60$, using as number of priorities $p \in \{2,\ \sqrt{n},\ n\}$ and *minimum* and *maximum* number of edges $(min,\ max) \in \left\{(1,\ n),\ \left(\frac{n}{2},\ n\right)\right\}$.

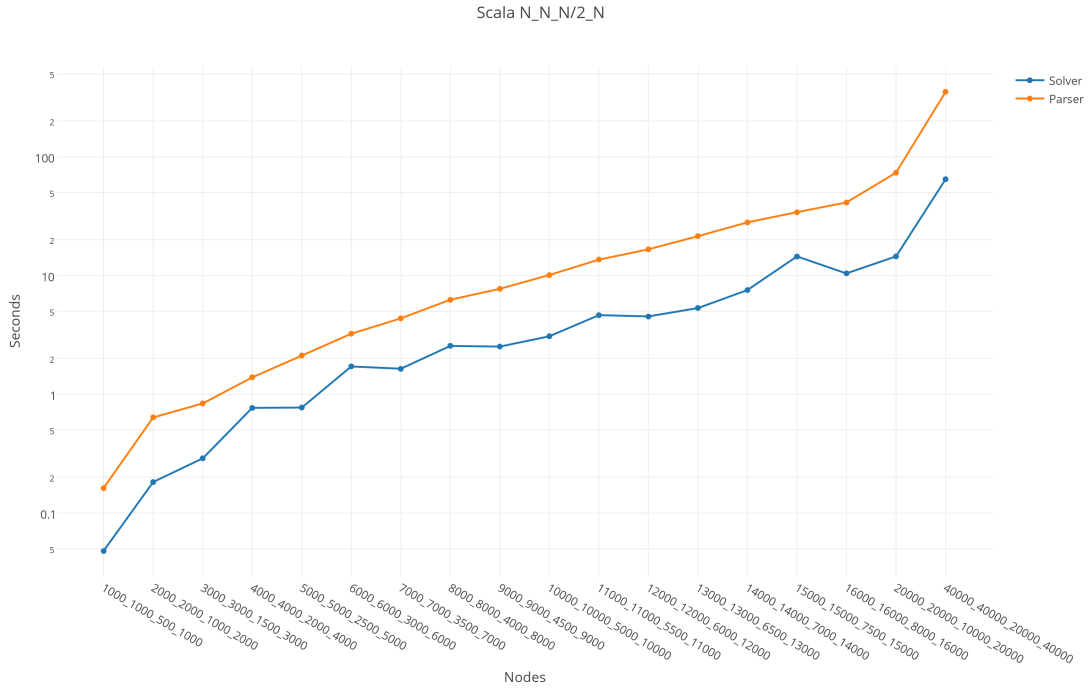

Figure 4.3.1: Scala N_N_N/2_N

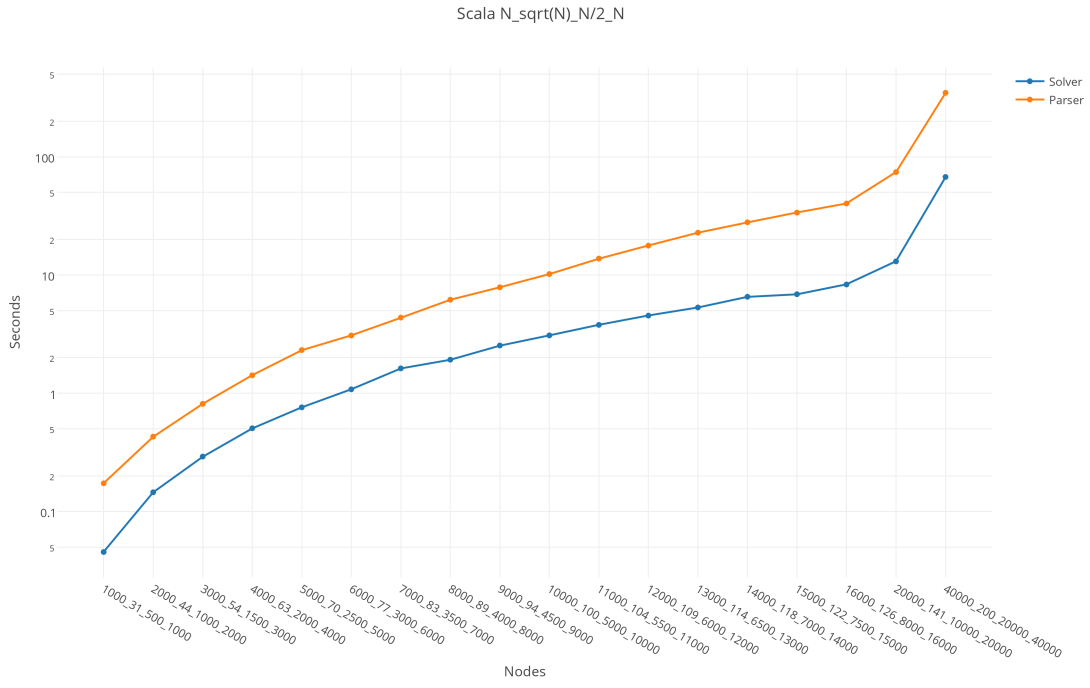

Figure 4.3.2: Scala N_sqrt(N)_N/2_N

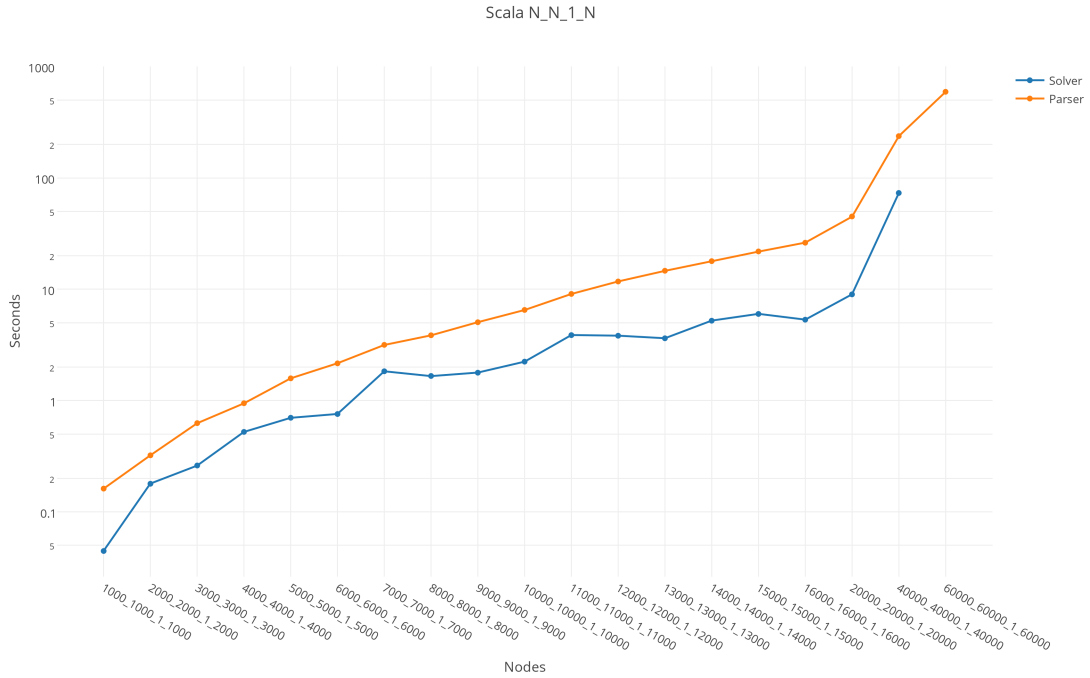

Figure 4.3.3: Scala N_N_1_N

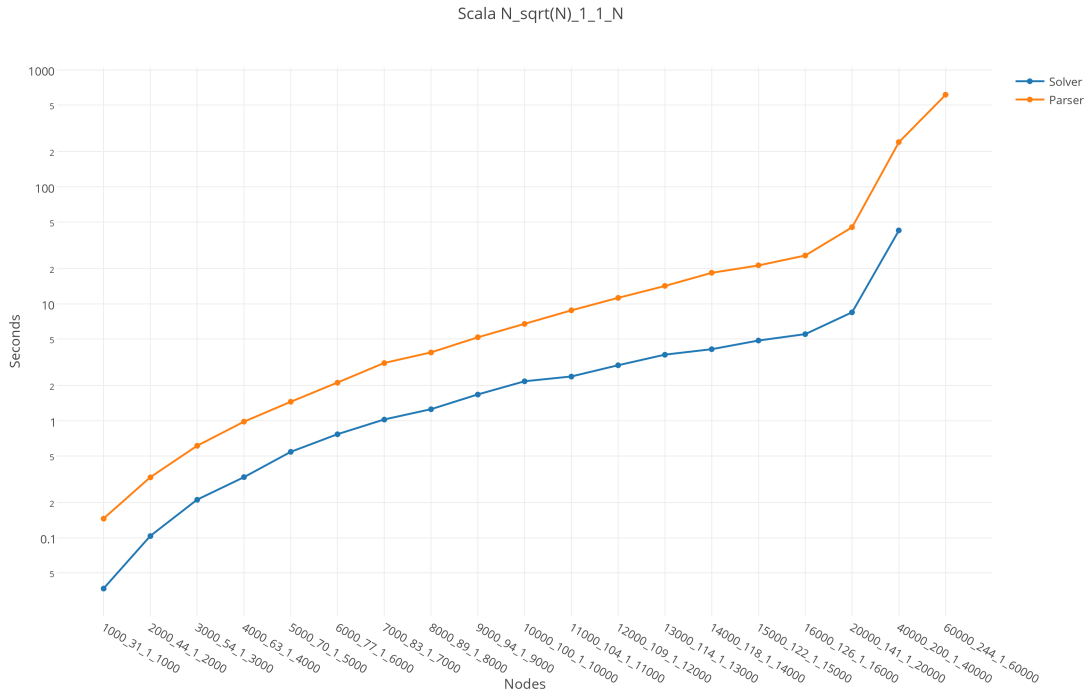

Figure 4.3.4: Scala N_sqrt(n)_1_N

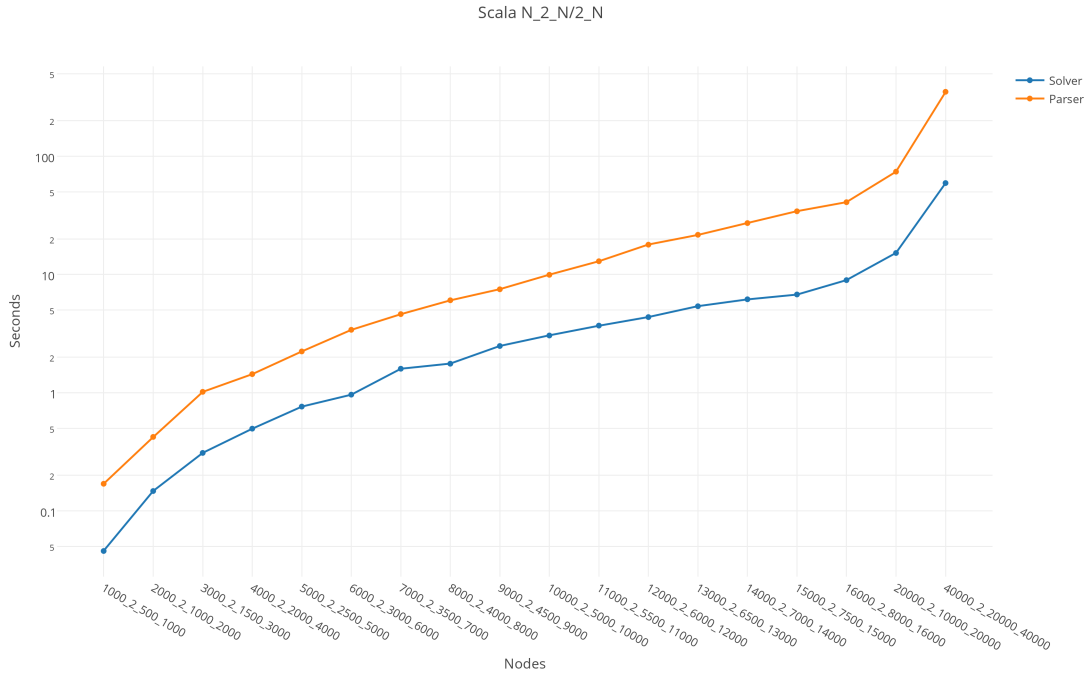

Figure 4.3.5: Scala N_2_N/2_N

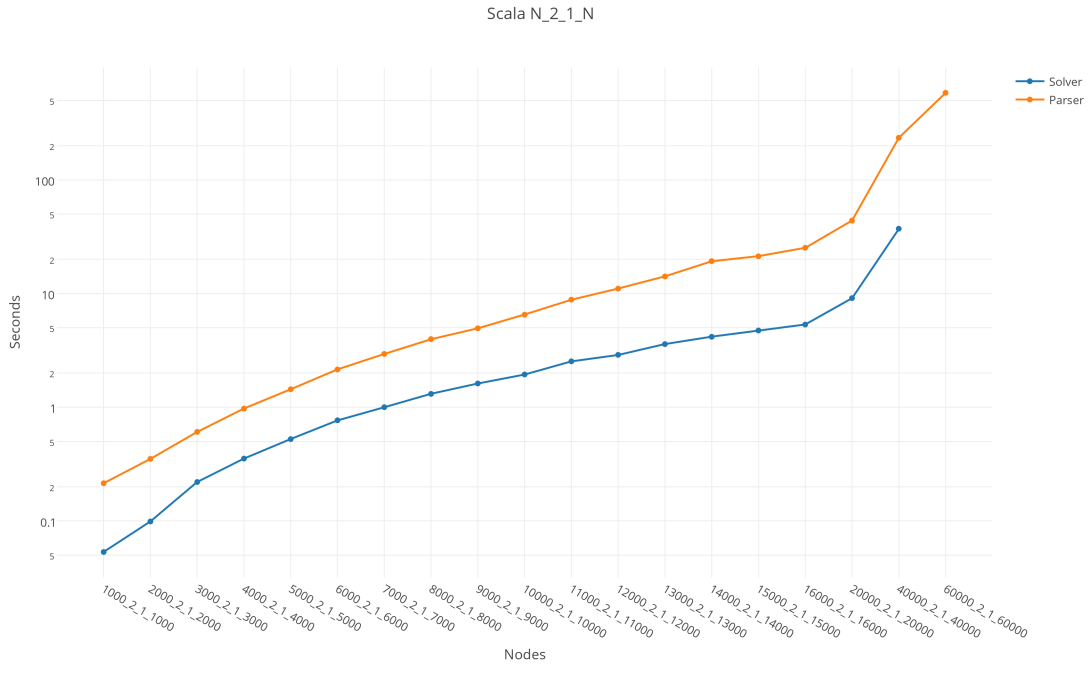

Figure 4.3.6: Scala N_2_1_N

5 Java

5.1 Introduction

The Java Programming Language [4] was developer by James Gosling at Sun Microsystem and introduces for the first time in 1995. It's one of the fastest growing and adopted programming technologies of all time. The designers of Java chose to use a combination of compilation and interpretation. A program written in Java is compiled into bytecode that will be fed to the *virtual machine*.

The language itself can be defined an opinionated languages, optimized for readability, simplicity for junior programmers, long term stability and robustness. These kind of design decision have come at a cost: verbosity and a type system that may seem inflexible compared to other languages. The original type system is 15 years old, simple and clear: types are references to classes, interfaces or primitive types. Classes are at the heart of the platform, all code that is intended to be executed must live inside a specific class. Primitives are all defined by the platform and developers are not allowed to define new primitives or extend from them. Interfaces are a very powerful way to abstract things, in a very object oriented programming, they cannot be instantiated directly and a class may implements an interface's API. Arrays hold either primitive types, instances of classes or even other arrays. Java refers its type system to a *strongly nominative typing* stating that every class must have a name by which it can be referred. Generics provide parameterized types, they are based on the idea that a type may act as a container for objects of another type.

5.1.1 The Java Virtual Machine

Historically, the first generation of the JVM was entirely an interpreter; nowadays the JVM uses a Just-In-Time (*JIT*) compiler [31, 1], a complex process aimed to improve performance at runtime. This process can be described in three steps: (1) source files are compiled by the Scala Compiler into Java Bytecode, that will be feed to a JVM; (2) the JVM will load the compiled classes at runtime and execute proper computation using an interpreter; (3) the JVM will analyze the application method calls and compile the bytecode into native machine code. This step is done in a lazy manner: the JIT compiles a code path when it knows that is about to be executed.

JIT removed the overhead of interpretation and allows programs to start up quickly, in addition this kind of compilation has to be fast to prevent influencing the actual performance of the program. Another interesting aspect of the JVM is that it verifies every class file after loading them. This makes sure that the execution step does not violate some defined safety properties. The checks are performed by the verifier that includes a complete type checking of the entire program. The JVM is also available on all major platforms and compiled Java executables can run on all of them with no need for recompilation.

5.2 Java 8

The latest release of Java is a huge step forward for the language that enriches the syntax and the standard library. It is a clear demonstration of a language evolution without compromising robustness, stability and still ensuring backward compatibility.

Interfaces can define static methods or have a *default* method. In the past it was impossible for a Java library to add methods to an interface without breaking existing code, this problem can be avoided using default methods. A new concept was introduced as *functional interfaces*. A functional interface is an interface that defines exactly one abstract method (i.e. *Runnable*), it was also introduced an annotation *@FunctionalInterface* that can be used to declare a interface as

functional. An extremely valuable addition are *lambdas*, a special expression that is auto-converted by the compiler to an instance of a class. The new Java *streams* provide utilities to support functional operations on streams of values. A stream can be seen as an iterator, the values flow and it can only be traversed once. Stream can be sequential or parallel, the operations can be *intermediate* or *terminal*. Terminal operations return a result of certain type, while intermediate operations return the stream itself letting the developer chaining multiple method calls in a single row. Intermediate operations are *lazy* and they are evaluated only when the terminal operation will be called. Java 8 contains also a brand new date and time API, repeatable annotations, method and constructor references, predicates, optionals, consumers, concurrent adders, the new Date API, parallel sorting, secure random generation, and a load of functional and concurrency features.

5.3 Implementation

The Java implementation relies mainly on the standard library, Google Guava [6] library and Trove [37] for high performing data structures. The Trove library provides high speed and memory efficient regular and primitive collections for Java. Internally Trove does not use any *java.lang.Number* subclasses, in this way there is no *boxing/unboxing* overhead. The *TIntArrayList* data structure is built on top of an array using the corresponding data type (*int[]* in this case). Each Trove Array List has several helper method inherited from the *java.util.Collections*. The resulting compiled *JAR*, that includes its dependencies, is around 7MB on disk.

Algorithm 5.1 Java Node Class

```
1  public static class Node {
2    private int index = -1;
3    private int player = -1;
4    private int priority = -1;
5    private final TIntArrayList adj = new TIntArrayList();
6    private final TIntArrayList inj = new TIntArrayList();
7
8    public int getIndex() { ... }
9    public void setIndex(int index) { ... }
10   public int getPlayer() { ... }
11   public void setPlayer(int player) { ... }
12   public int getPriority() { ... }
13   public void setPriority(int priority) { ... }
14   public void addAdj(int destination) { ... }
15   public void addInj(int origin) { ... }
16   public TIntArrayList getAdj() { ... }
17   public TIntArrayList getInj() { ... }
18 }
```

Algorithm 5.2 Java Graph Class

```
1  public class Graph {
2    public static Class Node { ... }
3    private final Node[] info;
4    public Graph(int numNodes) {
5        info = new Node[numNodes];
6        for (int i = 0; i < numNodes; i++) {
7            info[i] = new Node();
8        }
9    }
10   public int length() { ... }
11   public int getPlayerOf(int v) { ... }
12   public void addEdge(int origin, int destination) { ... }
13   public int maxPriority(BitSet removed) {
14     Optional<Node> maxNode = Stream.of(info)
15       .filter(x -> !removed.get(x.getIndex()))
16       .max((x, y) ->
17         Integer.compare(x.getPriority(), y.getPriority()));
18     return maxNode.isPresent() ? maxNode.get().getPriority() : -1;
19   }
20   public TIntArrayList getNodesWithPriority(final int pr,
21                                             BitSet rm) { ... }
22   public TIntArrayList incomingEdgesOf(int v) { ... }
23   public TIntArrayList outgoingEdgesOf(int v) { ... }
24   public static Graph initFromFile(String file) { ... }
```

Algorithm 5.3 Java Solver Interface

```
1 public interface Solver {
2     public int[][] win(Graph G);
3 }
```

Algorithm 5.4 Java RecursiveSolver Class

```
1 public class RecursiveSolver implements Solver {
2     @Override
3     public int[][] win(Graph G) {
4         BitSet removed = new BitSet(G.length());
5         return win_improved(G, removed);
6     }
7   ...
8 }
```

Algorithm 5.5 Java Attractor Function

```
1  private TIntArrayList
2  Attr(Graph G, TIntArrayList A, int i, BitSet removed) {
3    final int[] tmpMap = new int[G.length()];
4    TIntIterator it = A.iterator();
5    while (it.hasNext()) {
6      tmpMap[it.next()] = 1;
7    }
8    int index = 0;
9    while (index < A.size()) {
10     final TIntIterator iter = G.incomingEdgesOf(A.get(index)).iterator();
11     while(iter.hasNext()) {
12       int v0 = iter.next();
13       if (!removed.get(v0)) {
14         boolean flag = G.getPlayerOf(v0) == i;
15         if (tmpMap[v0] == 0) {
16           if (flag) {
17             A.add(v0);
18             tmpMap[v0] = 1;
19           } else {
20             int adjCounter = 0;
21             it = G.outgoingEdgesOf(v0).iterator();
22             while (it.hasNext()) {
23               if (!removed.get(it.next())) {
24                 adjCounter += 1;
25               }
26             }
27             tmpMap[v0] = adjCounter;
28             if (adjCounter == 1) {
29               A.add(v0);
30             }
31           }
32         } else if (!flag && tmpMap[v0] > 1) {
33           tmpMap[v0] -= 1;
34           if (tmpMap[v0] == 1) {
35             A.add(v0);
36           }
37         }
38       }
39     }
40     index += 1;
41   }
42   IntStream.of(A.toArray()).forEach(removed::flip);
43   return A;
44 }
```

Algorithm 5.6 Java Win Function

```
1  private int[][][]
2  win_improved(Graph G, BitSet removed) {
3    final int[][][] W = {new int[0], new int[0]};
4    final int d = G.maxPriority(removed);
5    if (d > -1) {
6      TIntArrayList U = G.getNodesWithPriority(d, removed);
7      final int p = d % 2;
8      final int j = 1 - p;
9      int[][][] W1;
10     BitSet removed1 = (BitSet)removed.clone();
11     final TIntArrayList A = Attr(G, U, p, removed1);
12     W1 = win_improved(G, removed1);
13     if (W1[j].length == 0) {
14       W[p] = Ints.concat(W1[p], A.toArray());
15     } else {
16       BitSet removed2 = (BitSet)removed.clone();
17       final TIntArrayList B =
18                     Attr(G, new TIntArrayList(W1[j]), j, removed2);
19       W1 = win_improved(G, removed2);
20       W[p] = W1[p];
21       W[j] = Ints.concat(W1[j], B.toArray());
22     }
23   }
24   return W;
25 }
```

5.4 Benchmarks

All tests in this section have been run on an Intel(R) i7(R) CPU @ 3.9GHz, with 32GB of Ram DDR3 1600Mhz. Precisely, we have used 20 random arenas generated through PGSolver of each of the following types, given $N = i \times 1000$ with i integer and $1 \leq i \leq 60$, using as number of priorities $p \in \{2,\ \sqrt{n},\ n\}$ and *minimum* and *maximum* number of edges $(min,\ max) \in \left\{(1,\ n),\ \left(\frac{n}{2},\ n\right)\right\}$.

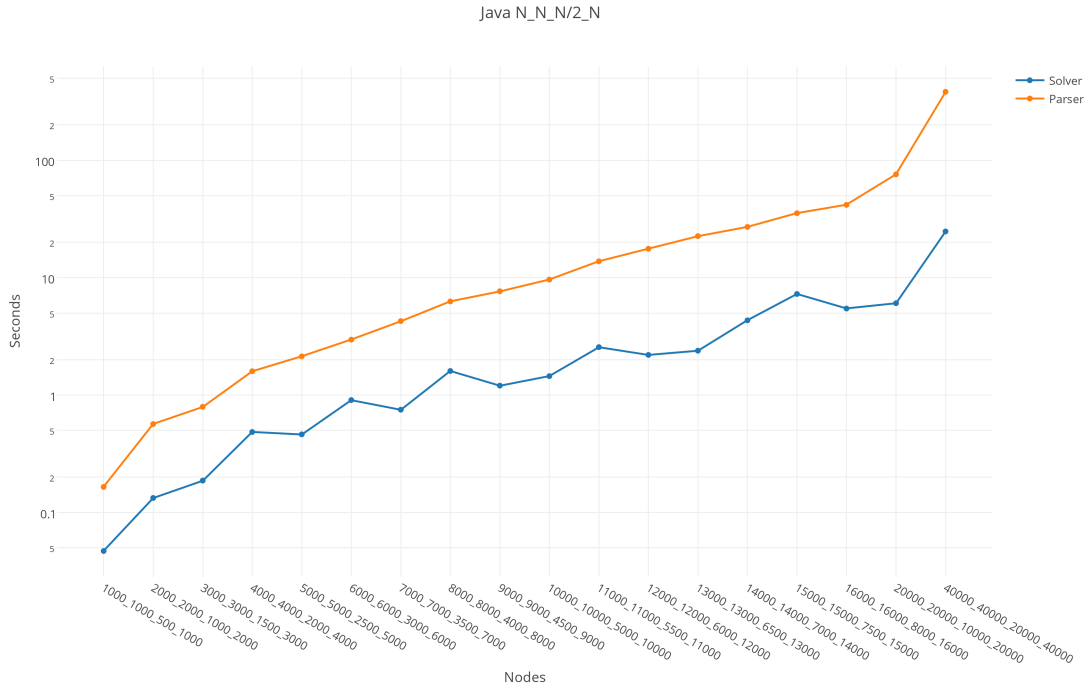

Figure 5.4.1: Java N_N_N/2_N

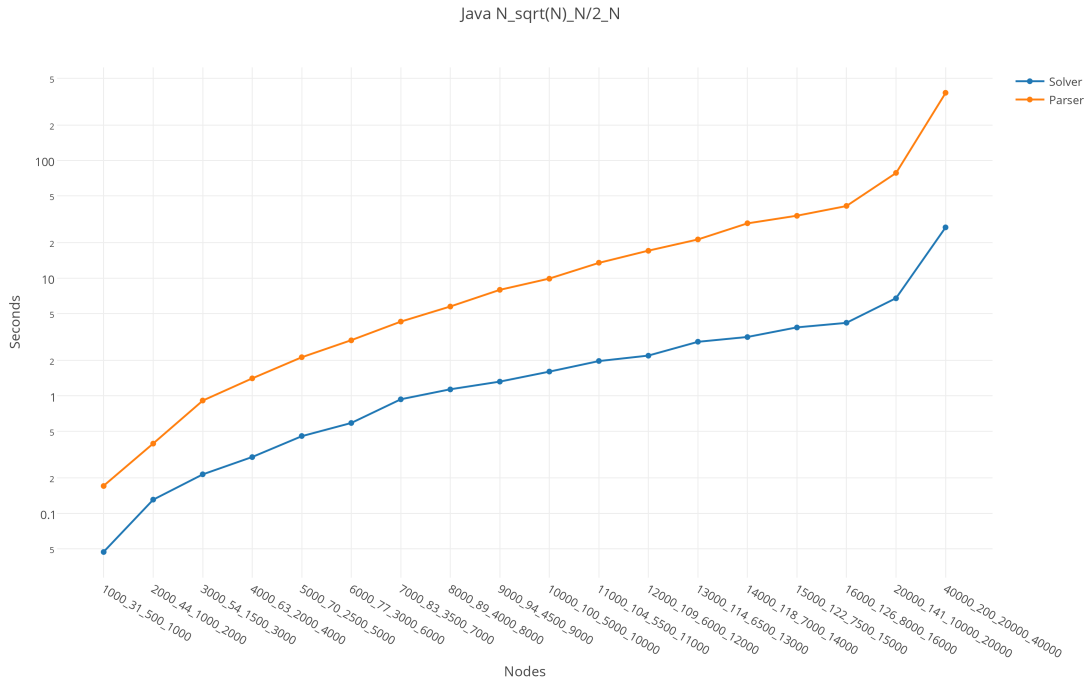

Figure 5.4.2: Java N_sqrt(N)_N/2_N

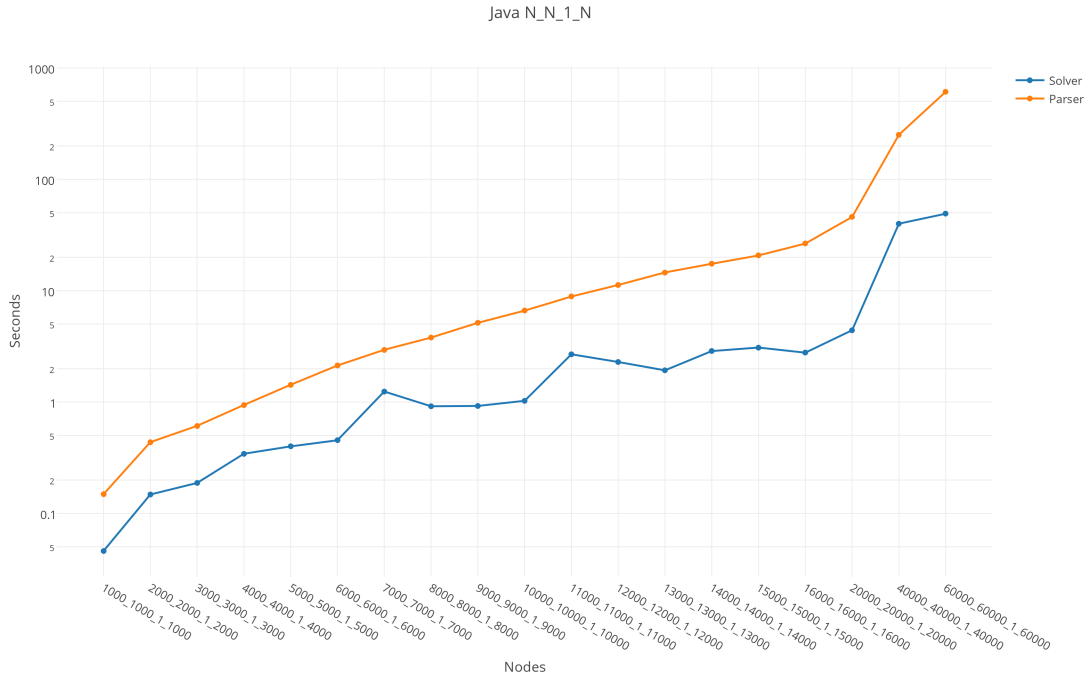

Figure 5.4.3: Java N_N_1_N

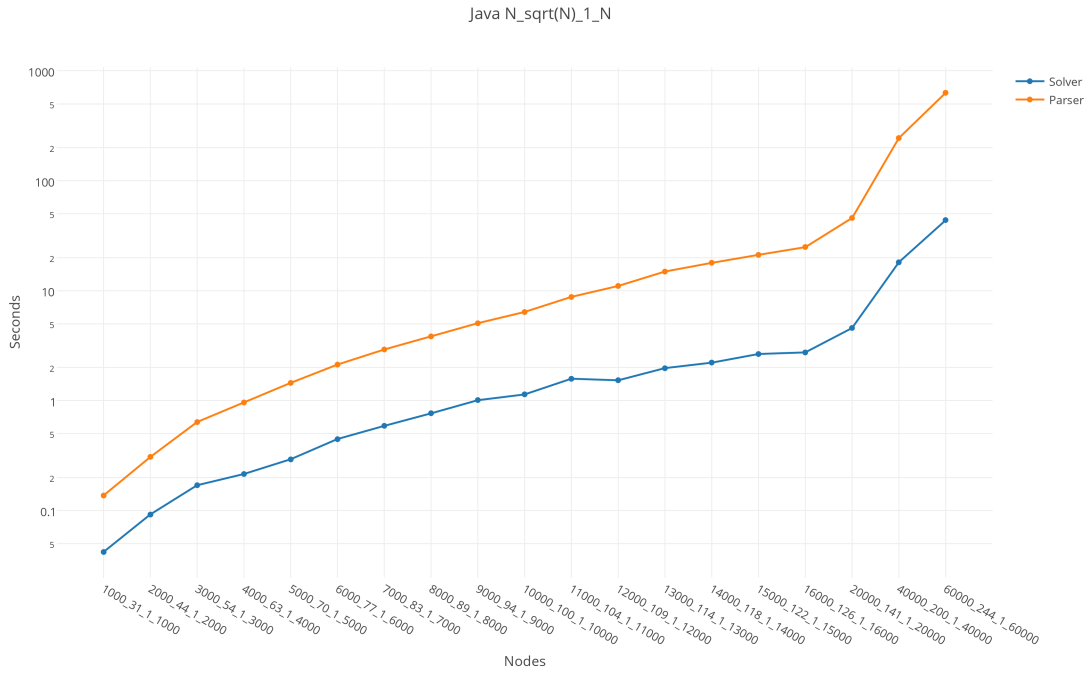

Figure 5.4.4: Java N_sqrt(n)_1_N

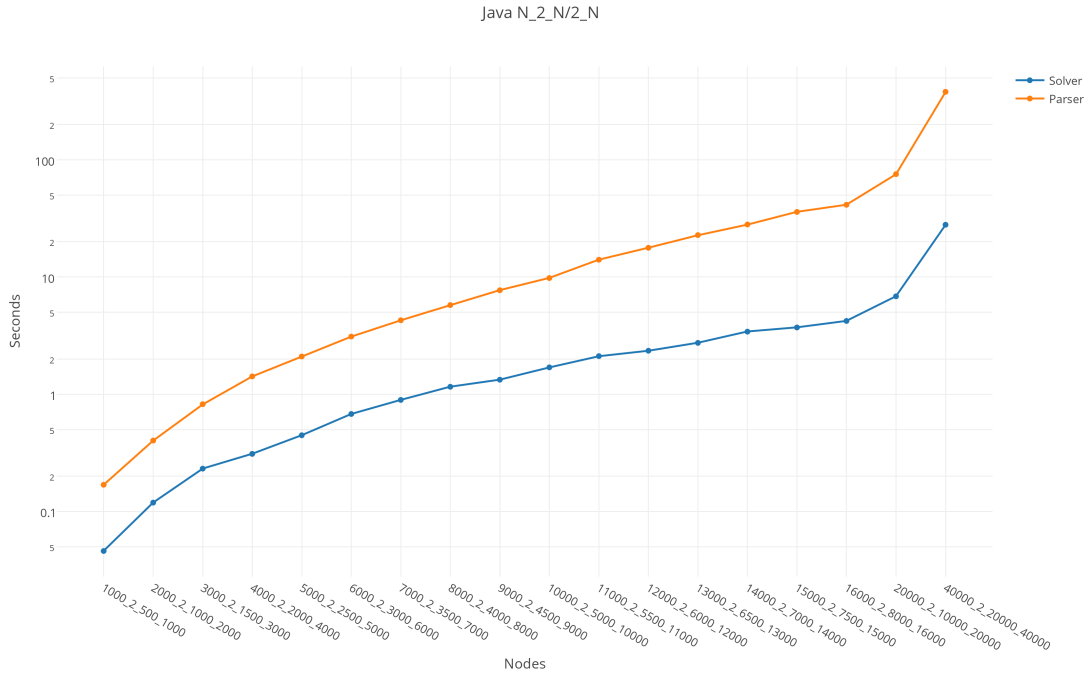

Figure 5.4.5: Java N_2_N/2_N

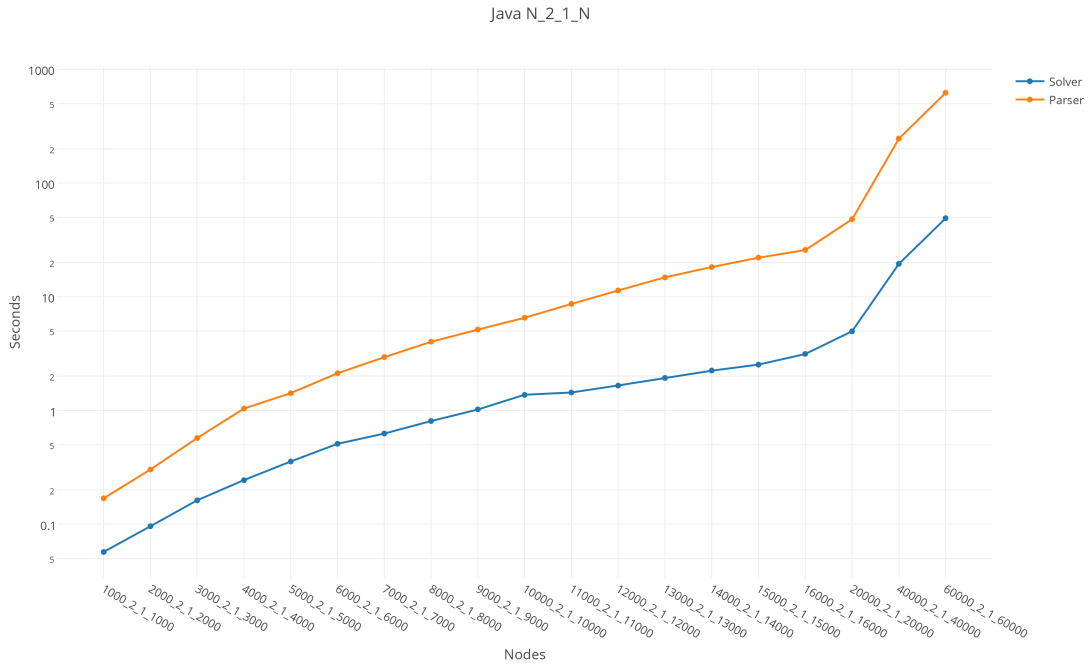

Figure 5.4.6: Java N_2_1_N

6 C++

6.1 Introduction

Through these years, C++ [36] has matured greatly, its focus, alongside the programming community, has widened from looking mostly at machine efficiency. The language itself was designed for system programming, focusing on performance, efficiency and flexibility as requirements. It has been used in a variety of context such as desktop applications, server side applications, databases, performance critical systems and video-games. The language is compiled, there are multiple compilers available such as *gcc* or *clang*. The standard is defined by an international organization (*ISO*) and many other programming languages have been influenced by C++, including Java, C# and C itself.

6.2 C++11

In 2011, the C++ standard committee released a major revision to the standard. This step in C++ evolution was focused to machine optimization techniques and programmer efficiency. Modern C++ can be seen in three parts: low level language inherited from C, advanced language features and the standard library (*stdlib)* that provides useful data structures and algorithms. C++11 [18] is the most recent version of the standard, extends the core standard library and includes several addition to the language core.

The new standard introduced the *auto* keyword introducing type inference, it can be seen as a placeholder for a type that tells the compiler to deduce the actual type of the variable; using auto is suggested and useful, aims to reduce the code

redundancy and gaining simplicity.

Another important topic is the threading facilities. For the first time, C++ standardized its concurrency model across multiple platforms. A thread class, *std::thread* is provided and for synchronization between threads, appropriate mutexes and condition variables are added to the library. In performance driven applications it is sometimes necessary to avoid the need of mutexes, this can be done using *std::atomic* library's lock-free operations on memory.

It is also important to underline the *RAII* (Resource Acquisition is Initialization) idiom. In RAII resource holding is tied to its lifetime, memory is allocated by the *constructor* and destroyed by the *destructor*. The use of RAII by every library, including the standard one, makes sure that all stack objects are destroyed at the end of the scope and therefore its use greatly simplifies memory management, overall code size and correctness.

Hash Tables *std::unordered_map*, regular expressions *std::regex*, smart pointers *std::unique_ptr* and *std::shared_ptr*, range-based for loops, lambda functions, object construction improvement, *nullptr* constant, strongly typed enumerations, *std::tuples* as collections of heterogeneous objects with fixed dimensions and so on, makes C++11 a milestone in C++ history, being the first major revision of the standard since 1998. The new edition underlines its continued importance as a general purpose programming language and valuable tool at the disposal of developers emphasizing programmer convenience, expressive power and increased performance.

6.3 Implementation

The solver implemented in C++ makes intense use of the language's standard library and the *Boost C++ libraries* [35] for string based algorithms when parsing files and timer functions. The tool was compiled with *clang [26],* a compiler front end for C, C++ and Objective-C that uses *LLVM [25]* as its backend. The compiled executable is around 62KB on disk and it must be noted that was compiled with full optimizations enabled (*-Ofast* flag). The code is C++11 compliant and

makes use of *std::vector* library to store integers or booleans. The C++ version, showed a huge memory footprint saving compared to garbage collected programming languages.

Algorithm 6.1 C++ Node Class

```
1  class Node {
2  private:
3      std::vector<int> adj;
4      std::vector<int> inj;
5      int priority;
6      int player;
7  public:
8      Node() {
9          priority = -1;
10         player = -1;
11     }
12     void set_priority(int pr) { ... }
13     void set_player(int pl) { ... }
14     int const get_priority() { ... }
15     int const get_player() { ... }
16     std::vector<int> const &get_adj() { ... }
17     std::vector<int> const &get_inj() { ... }
18     void add_adj(int other) { ... }
19     void add_inj(int other) { ... }
20 };
```

Algorithm 6.2 C++ Graph Class

```
1  class Graph {
2  private:
3      std::vector<Node> nodes;
4      std::map<int, std::vector<int>> priorityMap;
5  public:
6      Graph(int numNodes) {
7          nodes = std::vector<Node>(numNodes);
8      }
9      Node &get(long n) {
10         return nodes[n];
11     }
12     void addNode(int node, int priority, int player) { ... }
13     void addEdge(int origin, int destination) { ... }
14     long size() { ... }
15     std::map<int, std::vector<int>> &get_priority_map() { ... }
16 };
```

Algorithm 6.3 C++ Attractor Function

```
1  std::vector<int>
2  Attr(Graph& G, std::vector<bool>& removed, std::vector<int>& A, int i) {
3      std::vector<int> tmpMap(G.size(), -1);
4      for (const int x : A) {
5          tmpMap[x] = 0;
6      }
7      int index = 0;
8      while (index < A.size()) {
9          for (const int v0 : G.get(A[index]).get_inj()) {
10             if (!removed[v0]) {
11                 auto flag = G.get(v0).get_player() == i;
12                 if (tmpMap[v0] == -1) {
13                     if (flag) {
14                         A.push_back(v0);
15                         tmpMap[v0] = 0;
16                     } else {
17                         int adj_counter = -1;
18                         for (const int x : G.get(v0).get_adj()) {
19                             if (!removed[x]) {
20                                 adj_counter += 1;
21                             }
22                         }
23                         tmpMap[v0] = adj_counter;
24                         if (adj_counter == 0) {
25                             A.push_back(v0);
26                         }
27                     }
28                 } else if (!flag and tmpMap[v0] > 0) {
29                     tmpMap[v0] -= 1;
30                     if (tmpMap[v0] == 0) {
31                         A.push_back(v0);
32                     }
33                 }
34             }
35         }
36         index += 1;
37     }
38     return A;
39 }
```

Algorithm 6.4 C++ Win Function

```
std::array<std::vector<int>, 2>
win_improved(Graph& G, std::vector<bool>& removed) {
    std::array<std::vector<int>, 2> W;
    auto d = max_priority(G, removed);
    if (d > -1) {
        std::vector<int> U;
        for (const int x : G.get_priority_map()[d]) {
            if (!removed[x]) {
                U.push_back(x);
            }
        }
        int p = d % 2;
        int j = 1 - p;
        std::array<std::vector<int>, 2> W1;
        auto A = Attr(G, removed, U, p);
        std::vector<bool> removed1(removed);
        for (const int x : A) {
            removed1[x] = true;
        }
        W1 = win_improved(G, removed1);
        if (W1[j].size() == 0) {
            std::merge(W1[p].begin(), W1[p].end(), A.begin(), A.end(),
                       std::back_inserter(W[p]));
        } else {
            auto B = Attr(G, removed, W1[j], j);
            std::vector<bool> removed2(removed);
            for (const int x : B) {
                removed2[x] = true;
            }
            W1 = win_improved(G, removed2);
            W[p] = W1[p];
            std::merge(W1[j].begin(), W1[j].end(), B.begin(), B.end(),
                       std::back_inserter(W[j]));
        }
    }
    return W;
}
```

6.4 Benchmarks

All tests in this section have been run on an Intel(R) i7(R) CPU @ 3.9GHz, with 32GB of Ram DDR3 1600Mhz. Precisely, we have used 20 random arenas generated through PGSolver of each of the following types, given $N = i \times 1000$ with i integer and $1 \leq i \leq 60$, using as number of priorities $p \in \{2,\ \sqrt{n},\ n\}$ and *minimum* and *maximum* number of edges $(min,\ max) \in \left\{(1,\ n),\ \left(\frac{n}{2},\ n\right)\right\}$.

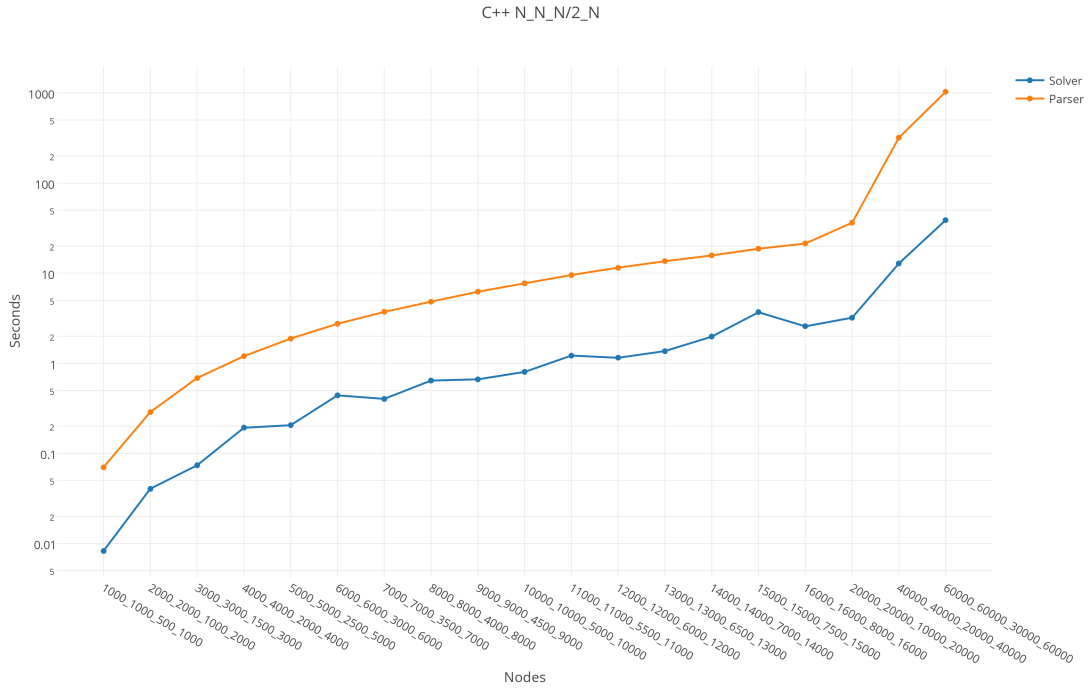

Figure 6.4.1: C++ N_N_N/2_N

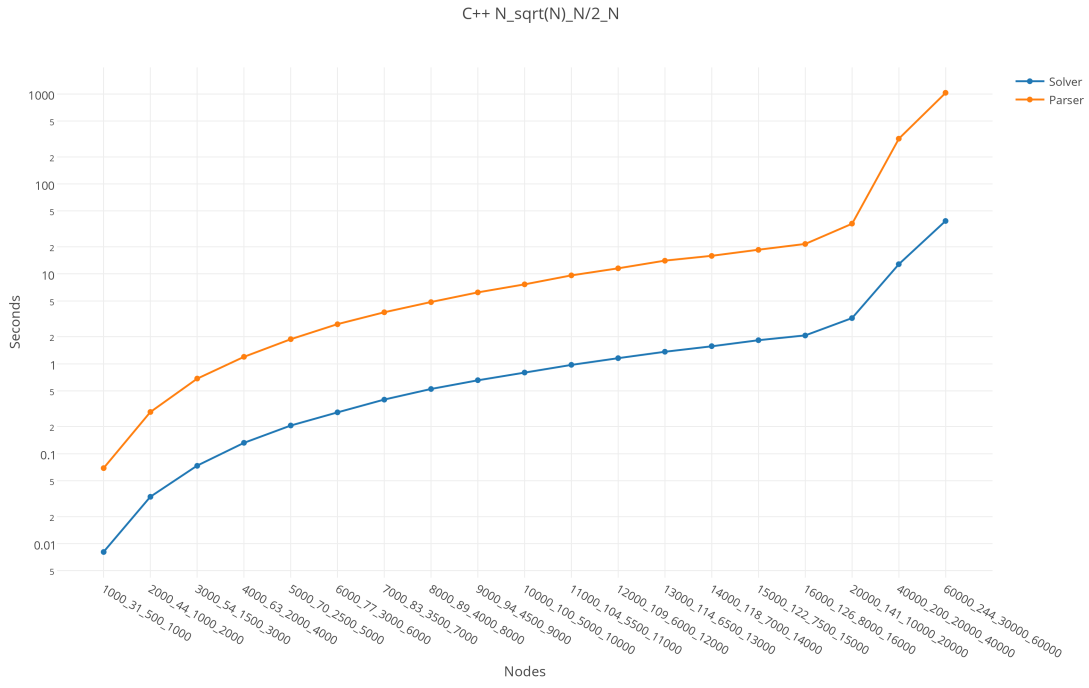

Figure 6.4.2: C++ N_sqrt(N)_N/2_N

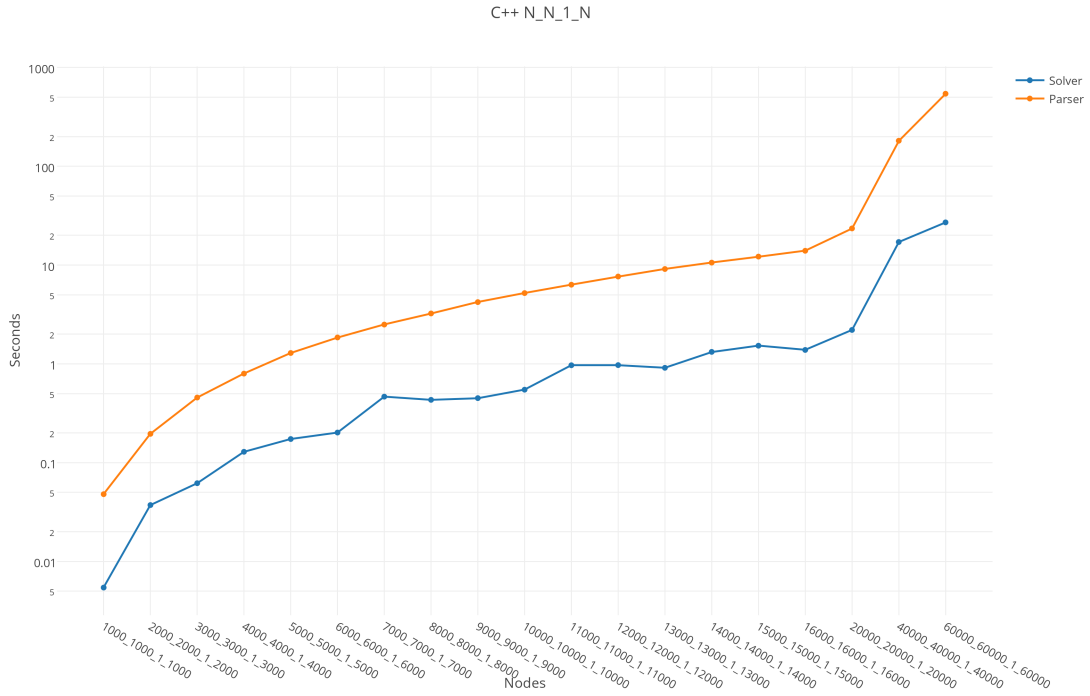

Figure 6.4.3: C++ N_N_1_N

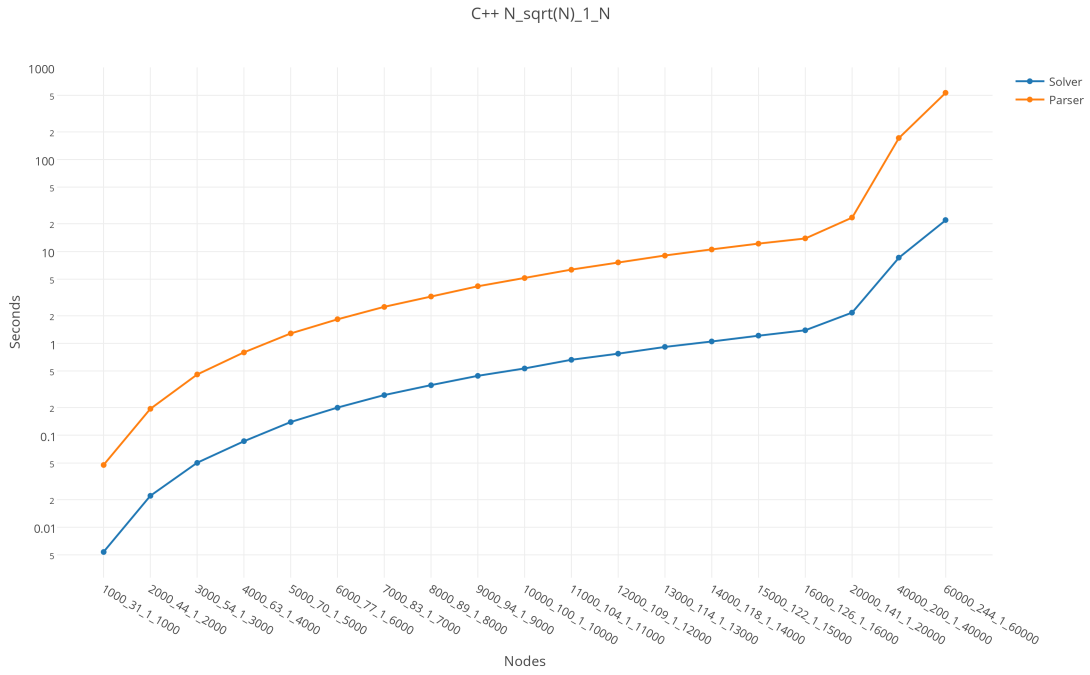

Figure 6.4.4: C++ N_sqrt(n)_1_N

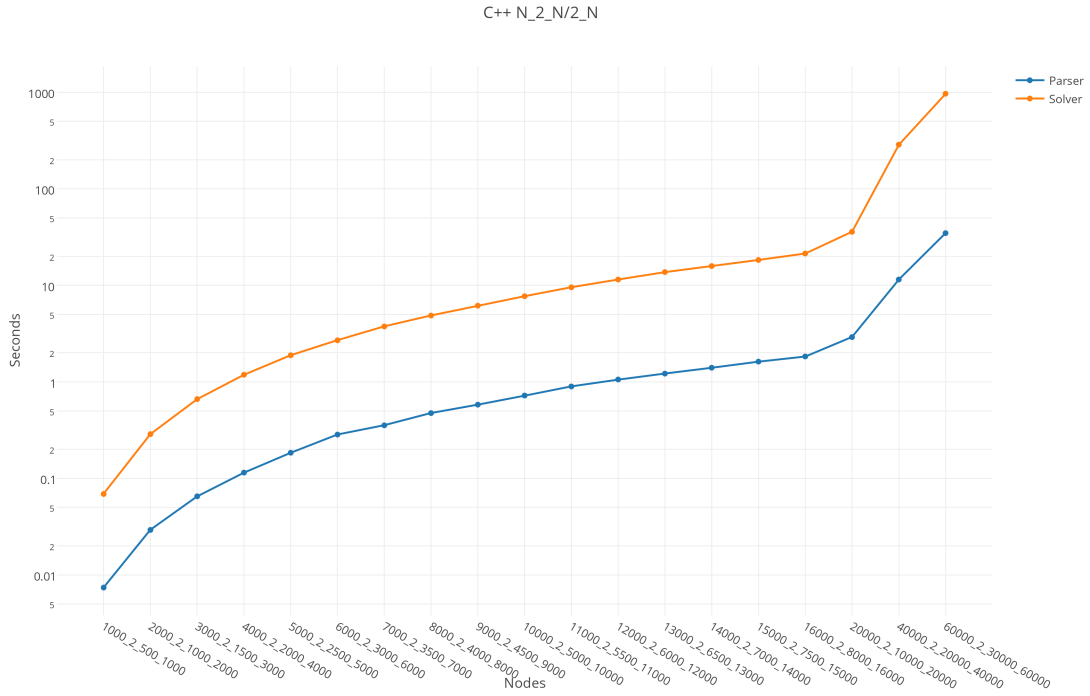

Figure 6.4.5: C++ N_2_N/2_N

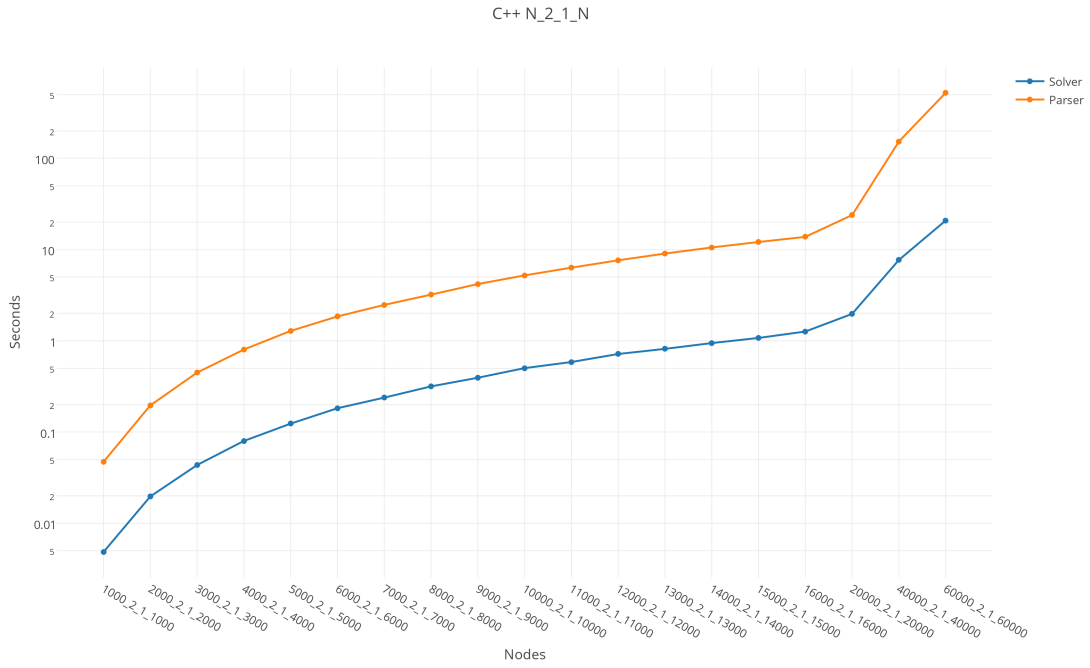

Figure 6.4.6: C++ N_2_1_N

6.5 C++ and Python

Python [39, 33] is a general purpose, high-level programming language. Its design is based on code readability and its syntax allows programmers to express concepts in few lines of code, "*One logical line of code equals one sentence in English*" [15]. *CPython* is the reference implementation of the standard, it is a bytecode interpreter written in C. The interpreter can be extended [38] with modules written in C or C++. The module here presented extends the Python interpreter it is written entirely in C++11, compiled with *clang* and includes *Boost.Python*, a framework for interfacing Python and C++ that allows to quickly and seamlessly expose C++ classes, methods or function to a Python program.

Algorithm 6.5 lists an interesting part of the module, excluding the classes and algorithms implementations that can be seen listed as Algorithm 6.1, 6.2, 6.3 and 6.4. It should be noted how quick and straightforward a developer can define a new Python module. The *BOOST_PYTHON_MODULE* macro is used to declare Python module initializations functions; it takes the *module name* as argument that needs to match the name of the module to initialized. Line 26 shows how it's easy to wrap C++ standard collections as Python generators, in this example the wrapped collection is a vector to integers; while line 22 defines a *__length__* method for a Graph object, in this way one can use in Python the builtin function *len()*.

Algorithm 6.5 C++ Module

```
#include <boost/python.hpp>
#include <boost/python/suite/indexing/vector_indexing_suite.hpp>
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
{...}
using namespace boost::python;

BOOST_PYTHON_MODULE(CPPSolver) {
   class_<Node, boost::noncopyable>("Node", boost::python::no_init)
   .def("set_priority", &Node::set_priority)
   .def("set_player", &Node::set_player)
   .def("get_priority", &Node::get_priority)
   .def("get_player", &Node::get_player)
   .def("add_adj", &Node::add_adj)
   .def("add_inj", &Node::add_inj)
   .def("get_adj", &Node::get_adj,
      return_value_policy<reference_existing_object>());

   class_<Graph, boost::noncopyable>("Graph", init<std::string>())
   .def(init<int>())
   .def("add_edge", &Graph::addEdge)
   .def("add_node", &Graph::addNode)
   .def("__len__", &Graph::size)
   .def("solve", &solve)
   .def("get", &Graph::get,
      return_value_policy<reference_existing_object>());

   class_<std::vector<int>>("IntVector")
   .def(vector_indexing_suite<std::vector<int>>());
}
```

7 Go

7.1 Introduction

The Go Programming Language [32] is an efficient, statically-typed compile language developed at Google in 2007 . It has built-in support for concurrency and communication, a latency-free garbage collection and high speed compilation process. The language itself follows the C tradition, but makes many changes aimed at conciseness, robustness and safety. The syntax is very similar to C, with some flavors of dynamic programming: for example there it's possible to declare a variable through *type inference* ($x := 0$ equals to $int \quad x = 0$). While the C programming language only supports procedural programming and Java forces a object-oriented paradigm (maybe less in Java 8), Go allows programmers to use the best paradigm for the problem; it can be used as a procedural language or as a object oriented one. Go provides a way to write system and server programs using concurrency primitives: light-weight processes, also called *goroutines*, *typed channels*, the *go* statement and the *select* statement that is a switch-like syntax to wait for communication over any set of channels. Go uses *interfaces*, a way to specify the behavior of an object as a set of methods. An interface can be satisfied (or implemented) by an arbitrary number of types, in languages like Java, implementing an interface must be explicit while in Go the implementation is automatically inferred at compile time. The *interface{}* type is the interface that specified an empty set of methods and it's satisfied by every value, so it can be used to accept *any* value.

The Go language designers aimed to create a programming language that would let to be productive without losing access to low level constructs. A kind of

balance achieved through deep research, a minimized set of keywords, builtin functions and clear syntax. The standard library provides all core packages programmers need to build real world programs, such as two fundamental built-in collection types: *slices* (variable-length arrays) and *maps* (key-value dictionaries or hashes), built to be efficient and to serve multiple purposes. The language does not hide pointers and there is no virtual machine getting in the way of performance, for this reason it's completely possible to design complex custom types with ease. The Go implementation for parity games, presented in this chapter, strictly follows the Go rules and conventions for code syntax and only requires the standard library to be compiled, keeping external dependencies at minimum.

The language itself was deigned to work well with server side applications and networking applications, in Go writing an *hello world* webserver is easy as printing the characters to the console, an example is listed as Algorithm 7.1. This simplicity is one of the most important features in the language, opens to a new area of enhancements of the solver itself, for example it may be interesting, as a future work, to write a web server for solving parity games and visualize the entire solving process, the results and graphs with a frontend application written in Javascript that interacts with the Go server.

Algorithm 7.1 Go Example

```
1  package main
2
3  import "net/http"
4
5  func main() {
6    http.HandleFunc("/", hello)
7    http.ListenAndServe(":8080", nil)
8  }
9
10 func hello(w http.ResponseWriter, r *http.Request) {
11   w.Write([]byte("Hello World!"))
12 }
```

Go is a fairly new programming language with strong key points accompanied with some weaknesses. The primary is the relatively small number of people using it as a general purpose language. Another missing feature is related to generics, this may be added in a future main release, the immaturity of the goroutine

scheduler. In addition, the garbage collector is fairly rudimentary, it's a *stop-the-world* mark-and-sweep, non-generational collector that periodically pauses applications when collection is running. This kind of behavior can be a critical issue for latency sensitive applications like games, audio processing, trading or enterprise systems.

7.2 Implementation

Algorithm 7.2 Go Node Struct

```
1  // Node struct keeps track of adjacent and incident lists.
2  // Its objects can be related to a vertex in a directed Graph
3  // data structure.
4  type node struct {
5    adj      []int
6    inj      []int
7    player   int
8    priority int
9  }
10
11 // Returns the node's Successors List
12 func (n *node) Adj() []int { ... }
13
14 // Returns the node's Predecessors List
15 func (n *node) Inj() []int { ... }
16
17 // Returns the node's Player
18 func (n *node) Player() int { ... }
19
20 // Returns the node's Priority
21 func (n *node) Priority() int { ... }
```

Algorithm 7.3 Go Graph Struct

```
1  // Graph is the main data structure, it has an array of 'node' objects
2  // and a map keeping track of every node with a given priority
3  type Graph struct {
4    priorityMap map[int][]int
5    nodes       []node
6  }
7
8  // Priorites returns the map of all priorities
9  func (g *Graph) Priorities() map[int][]int { ... }
10
11 // Nodes returns the nodes array
12 func (g *Graph) Nodes() []node { ... }
13
14 // NewGraph builds a new graph and preallocates all nodes up
15 // to the parameter 'numnodes' specified as input
16 func NewGraph(numnodes int) *Graph { ... }
17
18 // NewGraphFromPGSolverFile builds a new graph using a file, given as
19 // argument, that can be generated through PGSolver's generation tools
20 // or that respects the very same format.
21 func NewGraphFromPGSolverFile() *Graph { ... }
22
23 // MaxPriority returns the max priority in the graph
24 // takes a slice of booleans that will be excluded
25 // when looping through the graph
26 func (g *Graph) MaxPriority(removed []bool) int { ... }
27
28 // AddNode adds a node to the graph
29 func (g *Graph) AddNode(newnode int, priority int, player int) { ... }
30
31 // AddEdge adds a new edge to the graph
32 func (g *Graph) AddEdge(origin int, destination int) { ... }
```

Algorithm 7.4 Go Solver Interface

```
1  // Solver interface defines only a Win function
2  // that takes a graph as input and returns the solution
3  // for player 0 and player 1 as a tuple of integer slices
4  type Solver interface {
5    Win(graph *graphs.Graph) ([]int, []int)
6  }
7
8  // Solve function is exported letting users to
9  // extend the solvers of this tool using and adaptation
10 // of the well known Strategy Pattern in Go
11 func Solve(strategy Solver, graph *graphs.Graph) ([]int, []int) {
12   // Prints the time (in seconds) when the function returns
13   defer utils.TimeTrack(time.Now(), "Solved")
14   return strategy.Win(graph)
15 }
```

Algorithm 7.5 Go Attractor Function

```
1  // attr is the attractor function used by 'win'
2  func attr(G *graphs.Graph, removed []bool, A []int, i int) []int {
3    tmpMap := make([]int, len(G.Nodes()))
4    for _, x := range A {
5      tmpMap[x] = 1
6    }
7    index := 0
8    for {
9      for _, v0 := range G.Nodes()[A[index]].Inj() {
10       if !removed[v0] {
11         flag := G.Nodes()[v0].Player() == i
12         if tmpMap[v0] == 0 {
13           if flag {
14             A = append(A, v0)
15             tmpMap[v0] = 1
16           } else {
17             adj_counter := 0
18             for _, x := range G.Nodes()[v0].Adj() {
19               if !removed[x] {
20                 adj_counter += 1
21               }
22             }
23             tmpMap[v0] = adj_counter
24             if adj_counter == 1 {
25               A = append(A, v0)
26             }
27           }
28         } else if !flag && tmpMap[v0] > 1 {
29           tmpMap[v0] -= 1
30           if tmpMap[v0] == 1 {
31             A = append(A, v0)
32           }
33         }
34       }
35     }
36     index += 1
37     if index == len(A) {
38       break
39     }
40   }
41   return A
42 }
```

Algorithm 7.6 Go in Function

```
1  // win is the main function in Improved Zielonka's Recursive Algorithm
2  func win(G *graphs.Graph, removed []bool) ([]int, []int) {
3    var W [2][]int
4    d := G.MaxPriority(removed)
5    if d > -1 {
6      U := []int{}
7      for _, v := range G.Priorities()[d] {
8        if !removed[v] {
9          U = append(U, v)
10       }
11     }
12     p := d % 2
13     j := 1 - p
14     var W1 [2][]int
15     A := attr(G, removed, U, p)
16     removed1 := make([]bool, len(removed))
17     copy(removed1, removed)
18     for _, x := range A {
19       removed1[x] = true
20     }
21     W1[0], W1[1] = win(G, removed1)
22     if len(W1[j]) == 0 {
23       W[p] = append(W1[p], A...)
24     } else {
25       B := attr(G, removed, W1[j], j)
26       removed2 := make([]bool, len(removed))
27       copy(removed2, removed)
28       for _, x := range B {
29         removed2[x] = true
30       }
31       W1[0], W1[1] = win(G, removed2)
32       W[p] = W1[p]
33       W[j] = append(W1[j], B...)
34     }
35   }
36   return W[0], W[1]
37 }
```

Algorithm 7.7 Go Win exported Function

```
1  // Win is implemented by RecursiveImproved and returns
2  // the solution for a given game in input
3  func (r RecursiveImproved) Win(G *graphs.Graph) ([]int, []int) {
4    removed := make([]bool, len(G.Nodes()))
5    res1, res2 := win(G, removed)
6    return res1, res2
7  }
```

7.3 Benchmarks

All tests in this section have been run on an Intel(R) i7(R) CPU @ 3.9GHz, with 32GB of Ram DDR3 1600Mhz. Precisely, we have used 20 random arenas generated through PGSolver of each of the following types, given $N = i \times 1000$ with i integer and $1 \leq i \leq 60$, using as number of priorities $p \in \{2,\ \sqrt{n},\ n\}$ and *minimum* and *maximum* number of edges $(min,\ max) \in \left\{(1,\ n),\ \left(\frac{n}{2},\ n\right)\right\}$.

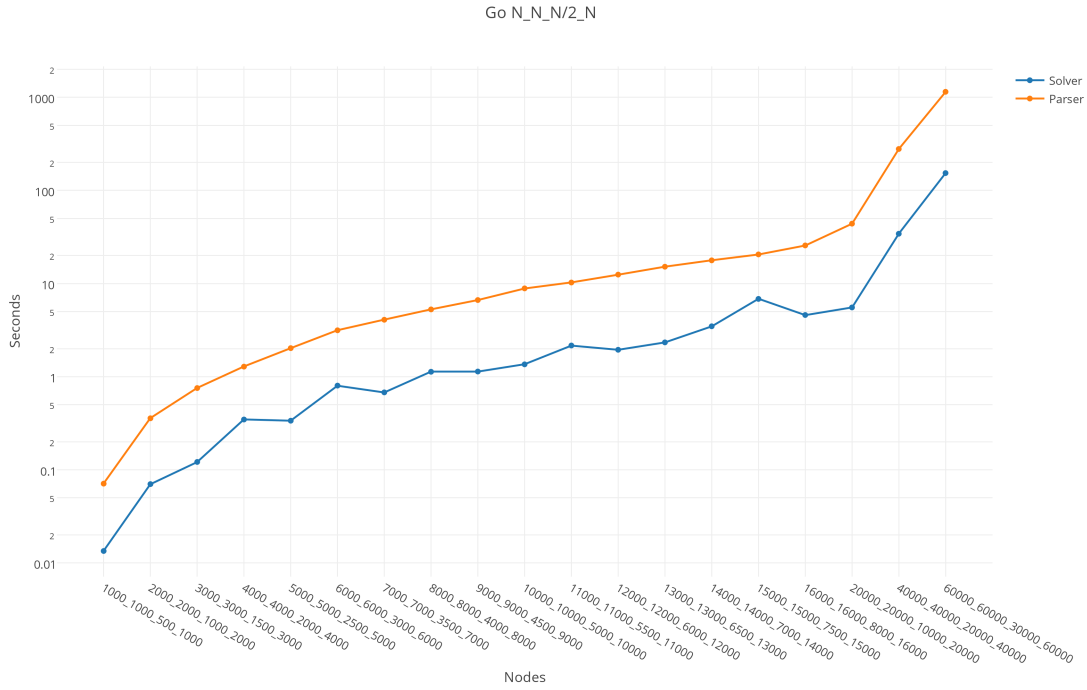

Figure 7.3.1: Go N_N_N/2_N

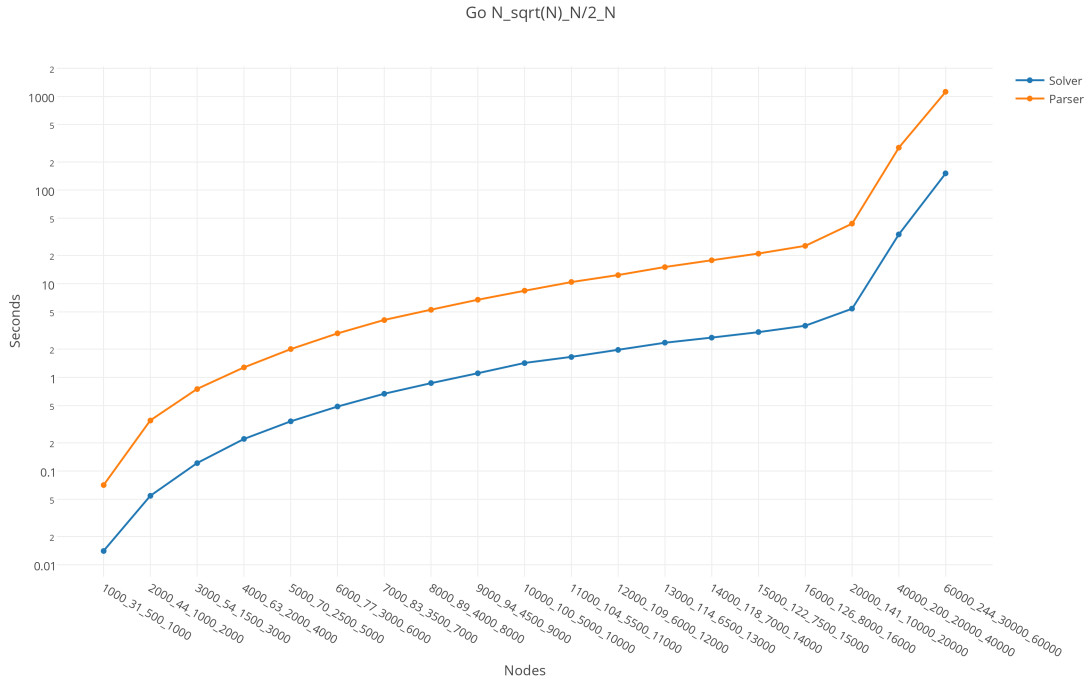

Figure 7.3.2: Go N_sqrt(N)_N/2_N

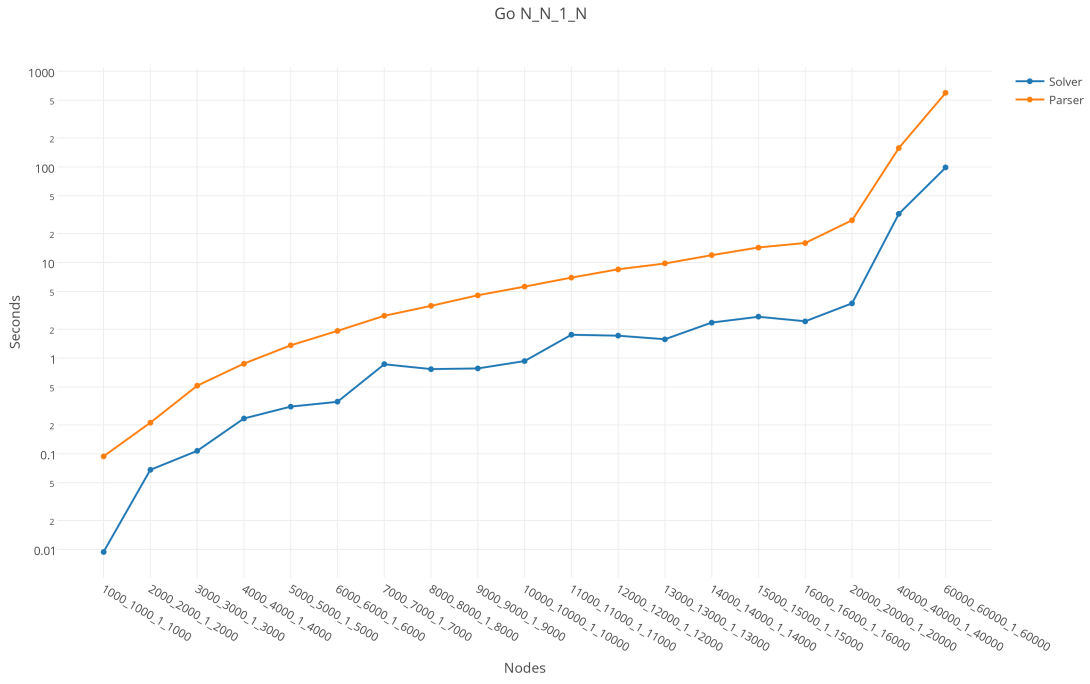

Figure 7.3.3: Go N_N_1_N

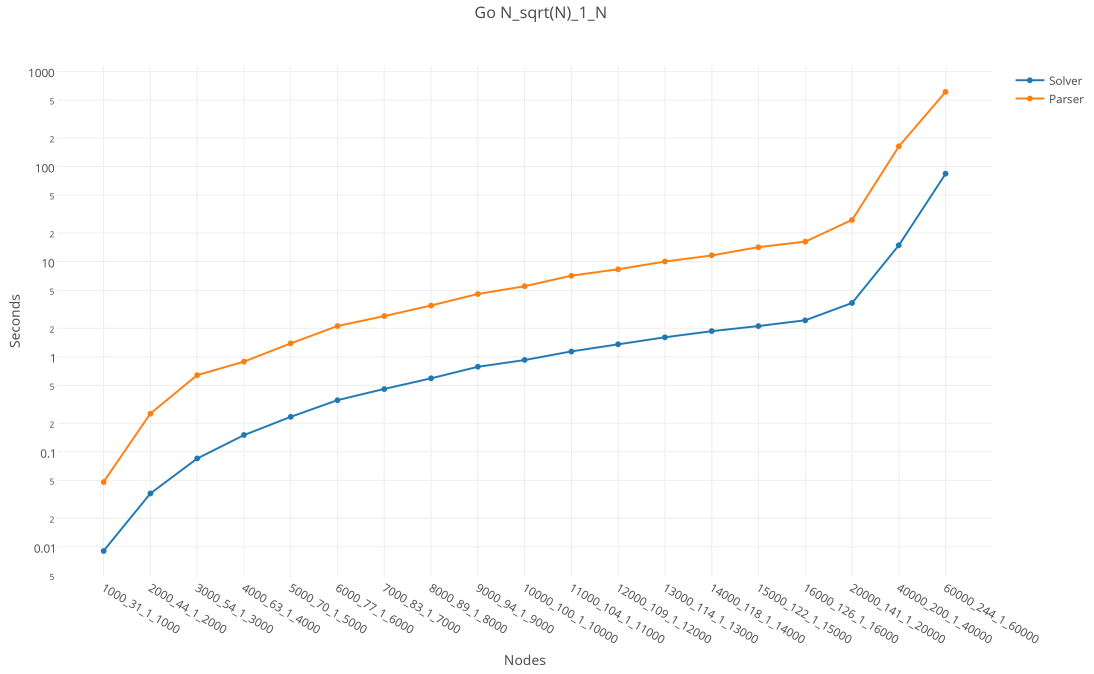

Figure 7.3.4: Go N_sqrt(n)_1_N

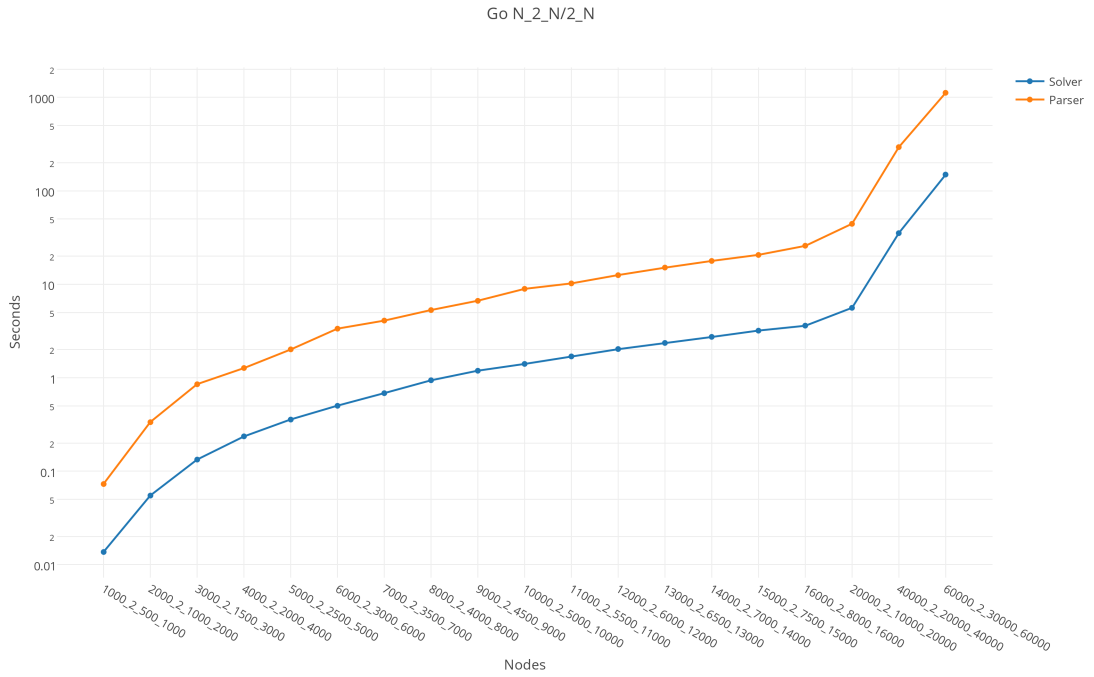

Figure 7.3.5: Go N_2_N/2_N

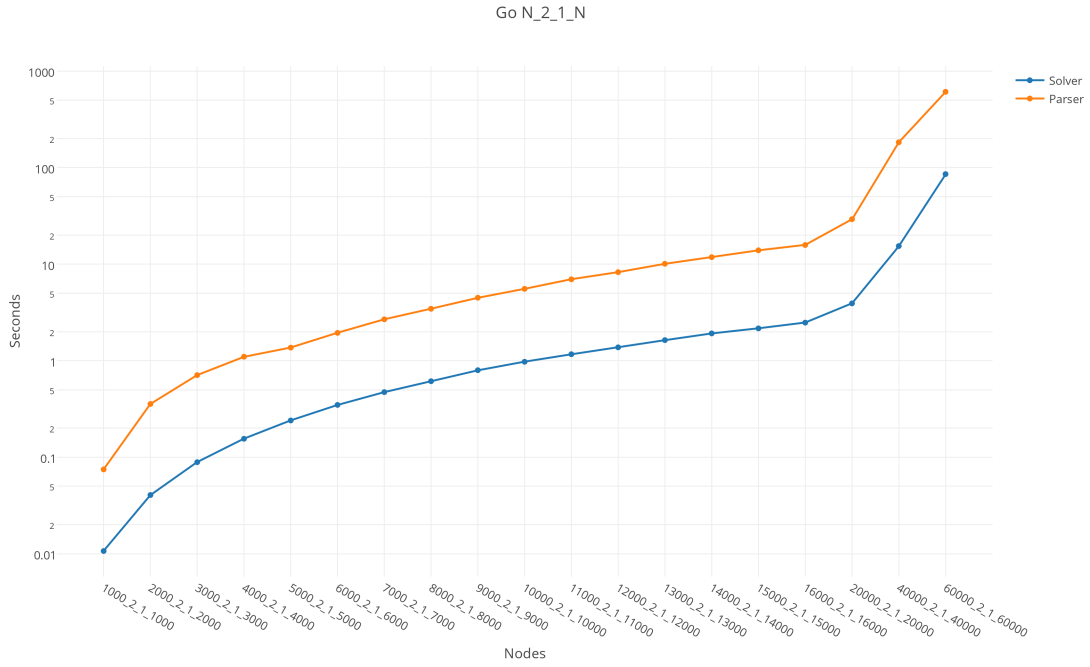

Figure 7.3.6: Go N_2_1_N

8 Comparison

The chosen languages have deep differences between each other, Go and C++ for example make use of a *static compiler* to produce a full native binary, while Java and Scala use the JVM's *JustInTime* compiler, also the JVM languages and Go are garbage-collected while C++11 uses *RAII* as a programming idiom where holding a resource is tied to an object lifetime and the language itself guarantees that an object is freed once control flow leaves the scope because of a return statement or an exception.

The following plots give a clear and perfect idea of which implementation is outperforming the solving process. C++ is the best performing one, in every test we have run, reducing the Scala running time by a factor of 4 and Go by a factor of ~1.5. Go's performance behavior tends to degrade after 40000 nodes, outperformed by Java in some cases. Scala and Java tend to have a very similar behavior over 40000 nodes, in fact in some plots the results are missing because the JVM would go over 32GB of available memory, while Go was always capable of completing the solving process even if taking more time. A further investigation on why performance was degrading so quickly for Go after the 40000 nodes threshold led to the garbage collector implementation, Go's collector is a *mark-and-sweep* with periodic pauses when it runs; on the other hand Java provides few different implementation for its collector allowing multiple performance optimizations (e.g. throughput vs latency).

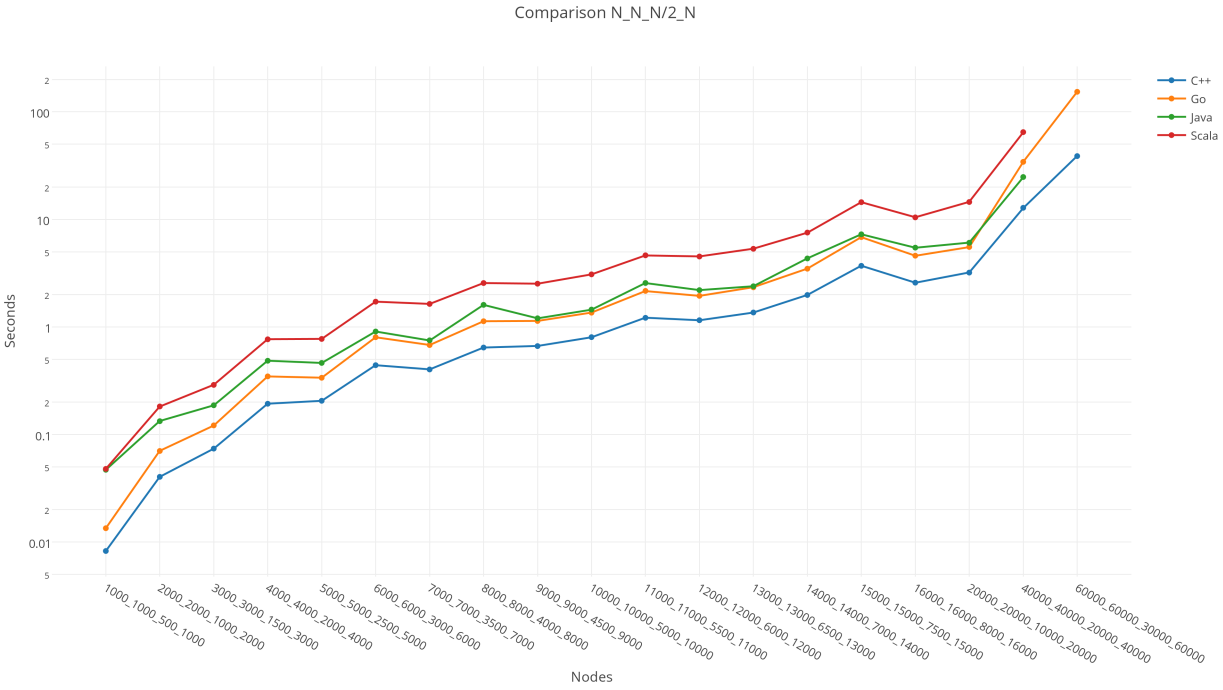

Figure 8.0.1: Comparison N_N_N/2_N

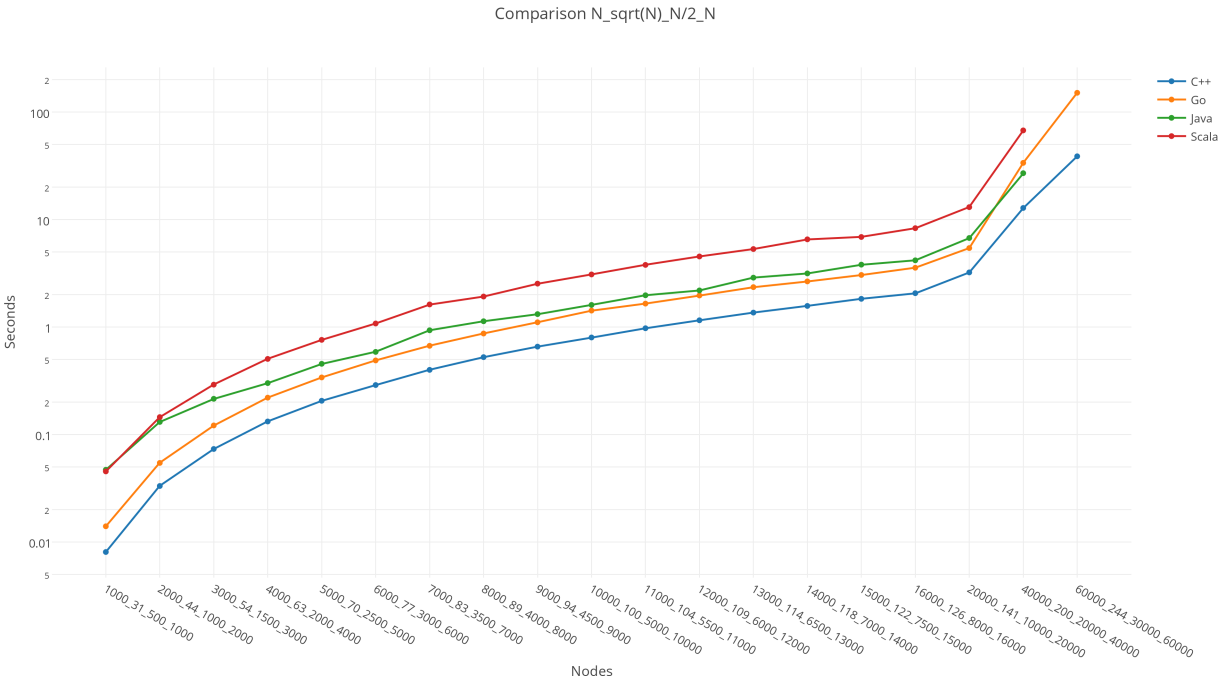

Figure 8.0.2: Comparison N_sqrt(N)_N/2_N

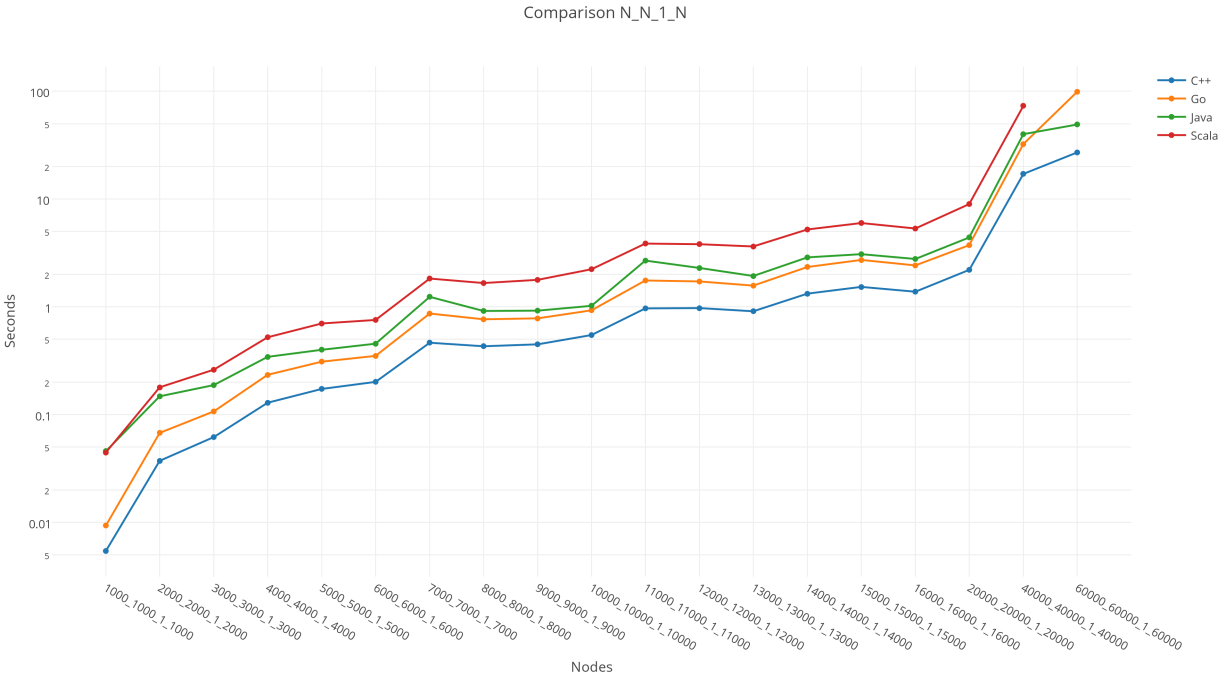

Figure 8.0.3: Comparison N_N_1_N

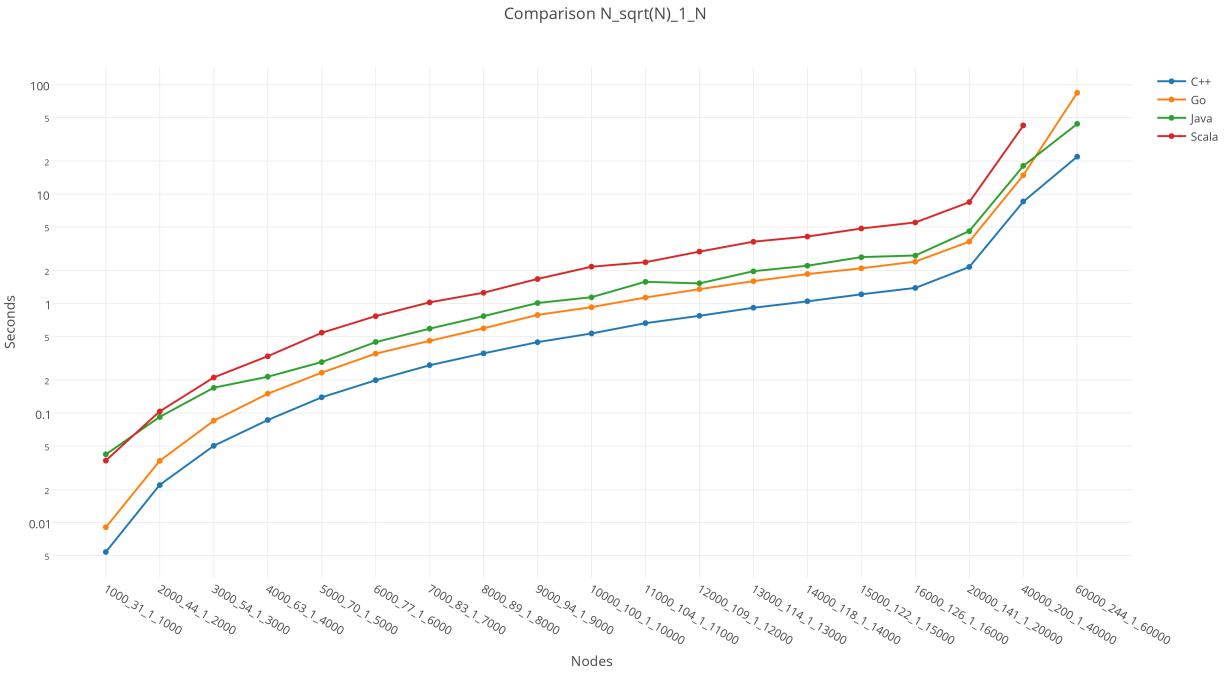

Figure 8.0.4: Comparison N_sqrt(n)_1_N

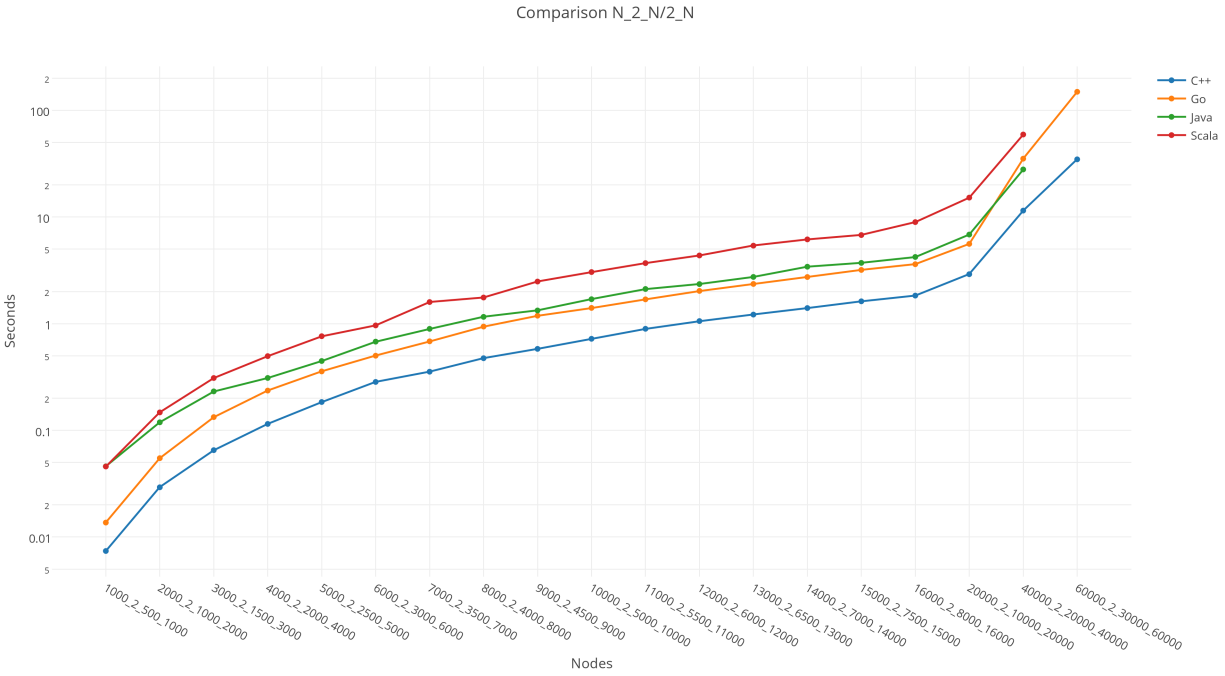

Figure 8.0.5: Comparison N_2_N/2_N

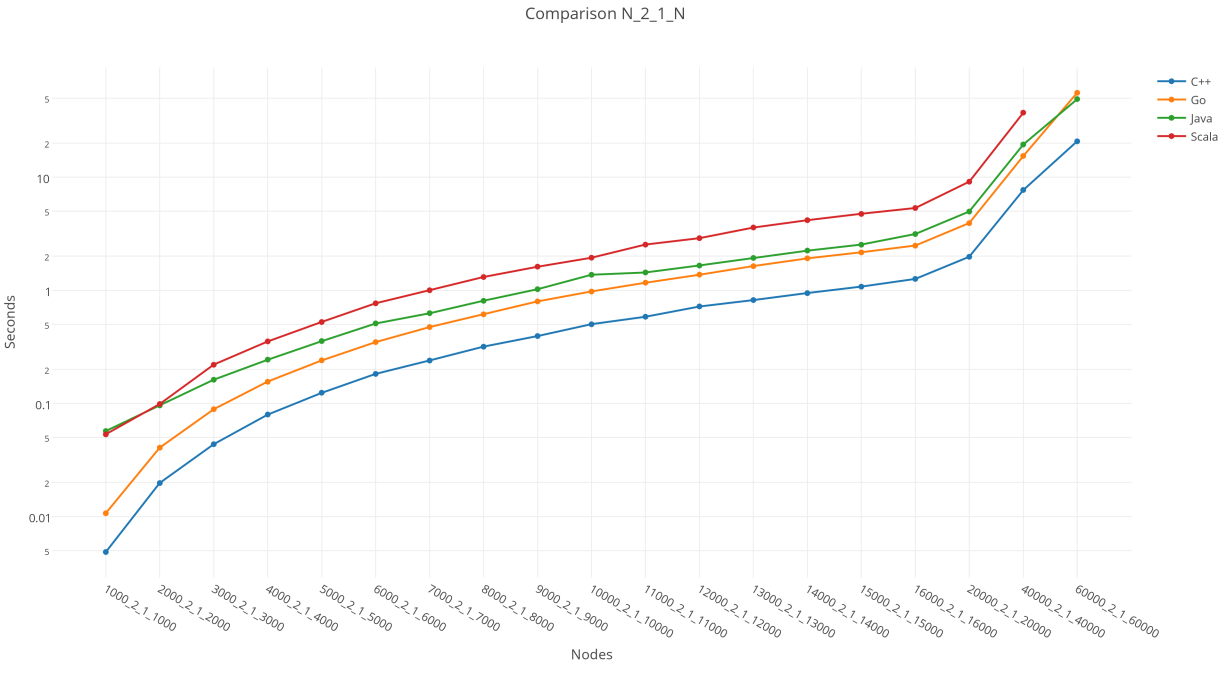

Figure 8.0.6: Comparison N_2_1_N

9 Conclusions

The main objective of this thesis was to improve the performance of the solving process of parity games. The importance of this kind of games is mostly because of the close relationship between them and *μ-calculus*. In other words, from a complexity point of view, solving a parity game is equivalent, under linear-time reductions to the introduced model checking problem, hence any tool for solving parity games is also a model checker for the *μ-calculus*. In formal system design and verification [9, 23], parity games arise as a natural evaluation machinery to automatically and exhaustively check for reliability of distributed and reactive systems [3, 5, 24].

Many techniques to improve the solving step were introduces in the past, for example we can refer the attempt by Philipp Hoffmann and Michael Luttenberger in [16], processing the entire system on a graphics processing units, or the many optimizations, perviously discussed as preprocessing steps, related to detection of special cases such as *self-cycle games*, *one-parity* game, *one-player* game, and priority compression and propagation. The step of priority compression attempts to reduce the number of priorities in a parity game, while the priority propagation aimed to increase priority but to reduce the range of priorities in a game and therefore compress the overall priorities.

PGSolver is a well-stablished framework that collects multiple algorithms to decide parity games. For several years now this platform has been the only one available to solve and benchmark in practice. A deep study was done against PGSolver's capabilities in solving parity game in an efficient and performant manner. In more details, even using the Zielonka's Recursive Algorithm, with SCC decompositions enabled PGSolver would require minutes to decide games

with few thousands of nodes, especially on dense graphs. The first step to improve the solving process was based on the work done in [10], where we introduced a slightly improved algorithm based on Zielonka's Recursive. The improved version guarantees that the original arena remains immutable tracking the removed nodes in every subsequent call and checking, in constant time, whether a node needs to be excluded or not. Casting this idea in the above automata reasoning, it is like enriching the state space with two flags (*removed*, $\neg$*removed*), instead of performing a complementation.

Another area where a similar approach can be exploited with good results is in the hierarchical state machines. One way to analyze a hierarchical machine is to flatten it, incurring in an exponential blow up, and apply model checking tools. The flattening technique can be avoided, as showed in [2]. Proceeding from this point, it would be useful to take a similar approach to the one used in [10], for example when working in a simple hierarchical state machine, a state may contain other machines, an example is showed in Figure 9.0.1.

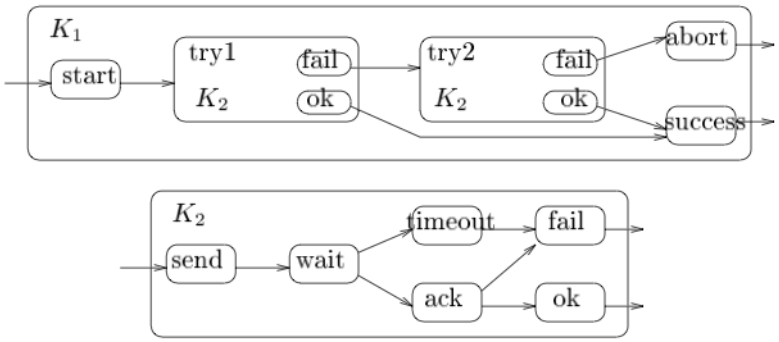

Figure 9.0.1: Example of an hierarchical state machine

We can easily imagine the process of resolving the state machine showed in the example as a function call with variables that define its state. An approach that can be taken is called *memoization*. In computing, this technique is an optimization used to speed up programs by storing the results of expensive function calls and returning the *cached* result when the very same inputs occur again. We can imagine that every machine in the example is a function call and the state of a function is set by its input parameters, using the memoization technique one can

implement and therefore reduce the need of an exponential blow up to solve this kind of problems: this is left as future work.

The plots presented in the sections above give a clear and perfect idea of which implementation is outperforming the solving process. C++ is the best performing one, in every test run, reducing Scala running time by a factor of 4 and Go by a factor of ~1.5. Go's performance behavior tends to degrade after 40000 nodes, outperformed by Java in some cases. The result of this work is the fastest implementation available in C++ for solving randomly generated parity games, and other multiple encodings that can be easily extended implementing new algorithms and raising several interesting questions. For example, what if one implements the other known algorithms to solve parity games? What if one implements a multithreaded version in every language? Can the result change for Go and Java versus C++? All these questions are left as future work.

Bibliography

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM SIGPlAN Notices*, volume 33, pages 280–290. ACM, 1998.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM SIGSOFT Software Engineering Notes*, 23(6):175–188, 1998.

[3] B. Aminof, F. Mogavero, and A. Murano. Synthesis of hierarchical systems. *Science of Comp. Program.*, 83:56–79, 2013.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 1996.

[5] B. Aminof and O. Kupferman and A. Murano. Improved model checking of hierarchical systems. *Inf. Comput.*, 210:68–86, 2012.

[6] B. Bejeck. *Getting Started with Google Guava.* Packt Publishing Ltd, 2013.

[7] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Notices*, volume 36, pages 114–124. ACM, 2001.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Oopsla 2002: Reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 48(4):46–57, 2013.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2002.

[10] A. Di Stasio, A. Murano, V. Prignano, and L. Sorrentino. Solving parity games in scala. *FACS 2014*, 2014.

[11] E. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS'91*, pages 368–377, 1991.

[12] O. Friedmann and M. Lange. The pgsolver collection of parity game solvers. *University of Munich*, 2009.

[13] O. Friedmann and M. Lange. Solving parity games in practice. In *ATVA'09*, LNCS 5799, pages 182–196. Springer, 2009.

[14] D. Gay and A. Aiken. *Memory management with explicit regions*, volume 33. ACM, 1998.

[15] R. Hettinger. Transforming code into beautiful, idiomatic python. PyCon, 2013.

[16] P. Hoffmann and M. Luttenberger. Solving parity games on the gpu. In *Automated Technology for Verification and Analysis*, pages 455–459. Springer, 2013.

[17] R. Hundt. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days*, 2011, 2011.

[18] I.S.O. Sc22/wg14 iso/iec 9899:2011. *Information technology - Programming languages - C*.

[19] M. Jurdzinski. Deciding the winner in parity games is in up ∩ co-up. *Inf. Process. Lett.*, 68(3):119–124, 1998.

[20] M. Jurdzinski. Small progress measures for solving parity games. In *STACS'00*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[21] M. Jurdzinski, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.

[22] D. Kozen. Results on the Propositional mu-Calculus. *TCS*, 27(3):333–354, 1983.

[23] O. Kupferman, M. Vardi, and P. Wolper. An Automata Theoretic Approach to Branching-Time Model Checking. *JACM*, 47(2):312–360, 2000.

[24] O. Kupferman, M. Vardi, and P. Wolper. Module Checking. *IC*,164(2):322–344, 2001.

[25] C. Lattner. Introduction to the llvm compiler system. In *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*, 2008.

[26] C. Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[27] A. Martin. Borel Determinacy. *AM*, 102(2):363–371, 1975.

[28] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[29] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004.

[30] M. Odersky, L. Spoon, and B. Venners. *Programming in scala*. Artima Inc, 2008.

[31] M. Paleczny, C. Vick, and C. Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[32] R. Pike. The go programming language. *Google Tech Talks*, 2009.

[33] G. v. Rossum et al. Python programming language. *python.org*, 1989.

[34] S. Schewe. Solving parity games in big steps. In *FSTTCS'07*, LNCS 4855, pages 449–460. Springer, 2007.

[35] B. Schling. *The boost C++ libraries*. Xml Press, 2011.

[36] B. Stroustrup et al. *The C++ programming language*. Pearson Education India, 1995.

[37] G. Trove. High performance collections for java.

[38] G. Van Rossum and F. L. Drake. *Extending and embedding the Python interpreter*. Centrum voor Wiskunde en Informatica, 1995.

[39] G. Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, 2007.

[40] J. Vöge and M. Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *CAV'00*, LNCS 1855, pages 202–215. Springer, 2000.

[41] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.