Android Genetic Programming Framework

Alban Cotillon, Philip Valencia, and Raja Jurdak

Autonomous Systems Laboratory
CSIRO ICT Centre, Brisbane Australia
alban.cotillon@insalien.org, { philip.valencia, raja.jurdak } @csiro.au

Abstract. Personalisation in smart phones requires adaptability to dynamic context based on application usage and sensor inputs. Current personalisation approaches do not provide sufficient adaptability to dynamic and unexpected context. This paper introduces the Android Genetic Programming Framework (AGP) as a personalisation method for smart phones. AGP considers the specific design challenges of smart phones, such as resource limitation and constrained programming environments. We demonstrate AGP's utility through empirical experiments on two applications: a news reader application and an energy efficient localisation application. Results show that AGP successfully adapts application behaviour to user context.

1 Introduction

Smartphones have experienced exponential growth in recent years. These phones embed a growing diversity of sensors, such as gyroscope, accelerometer, Global Positioning System (GPS), and cameras, with broad applicability in areas such as urban sensing or environmental monitoring. Coupled with diverse user profiles [1], this provides significant user personalization opportunities, such as location-based and usage-based services, but it also involves significant challenges in adaptation to new or unexpected context.

Most smartphone algorithms, aiming at either data-centric [4] or user-centric personalization [2], are based on static or rule-based approaches. However, personalization increasingly depends on contextual information and user inputs [3]. Both are subject to dynamic changes, which motivates the use of methods that can not only adapt to expected changes or behaviors, but can also learn how to deal with unexpected changes in context.

Online learning is well-suited for smart phone personalization. In particular, online genetic programming supplies common basic constructs for a smart phone application that can evolve over time according to individual user preferences. This paper introduces Android Genetic Programming Framework (AGP) for the mobile Operating System (OS), Android. We have chosen the Android platform as it is mainly open-source. As far as we know, this is the first genetic programming solution available on smartphones. The AGP framework can deal with dynamic fitness functions, providing a context-specific solution. Our goal is to demonstrate the flexibility of our new platform and how it can solve multi-objective problems in a dynamic environment.

2 Related Work

Several previous works in Genetic Programming have focused on architectural issues. Whereas some solutions provide generic frameworks for Evolutionary Computation problems [6,8,9], others propose application-specific solutions. Ismail et al. [10] describe a GP framework for extracting a mathematic formula needed for Fingerprint Matching, whereas, in Pattern Recognition [11], the authors focus on a genetic programming framework for content-based image retrieval. In Learning to Advertise [14], Lacerda et al. introduce a framework for associating ads with web pages based on GP. Valencia et al. [12] study genetic programming for Wireless Sensor Networks and propose the In Situ Distributed Genetic Programming (IDGP) framework. DGPF [15] brings utilities for Master/Slave, Peer-to-Peer, and P2P/MS hybrid distributed search execution. P-Cage [22] introduces and evaluates a complete framework for the execution of genetic programs in a P2P environment. It shows the relevance of using P2P networks scalability to counteract computation limitations. GA implementation has already been done on portable devices such as the Nokia N73 [7]. Its reliance on Python requires the user to install Python Runtime and various libraries whereas AGP can be used off-the-shelf without any additional module.

Design patterns describe the interaction between groups of classes or objects. They concentrate on specific concerns for implementing source code to support program organization. When they are well integrated into a framework, they ensure the goals of extensibility and reuse. Lenaerts and Manderick [13] discuss the construction of an object-oriented Genetic Programming framework using on design patterns to increase flexibility and reusability. McPhee et al. [8] extend the latter to Evolutionary Computation (EC). As the problem to solve becomes wider, it leads to a more abstract set of classes. Based on those works, Ventura et al. introduced JCLEC [6], a Java Framework for evolutionary computation. They present a layered architecture and provide a GUI for EC. This paper similarly uses Java for a genetic programming framework, albeit for a more resource constrained smart phone platform.

3 AGP Framework: Android Genetic Programming Framework

3.1 Motivation

Mobile phone users have always tried to customise their devices, for instance through personalised ring tones. The emergence of smartphones takes the personalization possibilities to a new level. First of all, smartphones have access to a huge diversity of Internet data which can be augmented with sensor-based context information. Secondly, the higher computing performance of smart phones enables developers to create novel applications. The combination of processing power, content accessibility and context awareness opens new opportunities for personalization. However, personalisation mechanisms have been slow to respond

to these opportunities, relying hugely on static or rule-based algorithms. These approaches suffer in adaptability to unexpected changes, which requires a shift in personalisation methods.

Online genetic programming can evolve over time and is able to gather data from different sources. Because it continuously assesses new application configurations, genetic programming can improve the user experience by incorporating data from multiple sensors to tailor application performance to user preferences.

3.2 Design challenges

Android is an operating system for mobile devices developed and maintained by Google and the open-source community. It provides an API which allows programmers to develop applications using the Java language. We used this particular API to implement the AGP framework. Although several other GP frameworks are available for the Java platform, none is suitable for Android because Android replaces several subsets of Java class libraries from the JavaSE with its own new classes. Moreover, developing a GP framework for a mobile OS brings some challenges that are discussed in this section.

User interaction GP frameworks designed for desktop machines focus on solving of complex problems offline. On the other hand, Android smartphones provide an opportunity for online learning of user-specific context to improve user experience. These devices provide increasingly sophisticated applications that can be customised to user preferences. To allow the capture of this diversity richness, the AGP framework directly accesses the Android API functions to learn from the user context without requiring a dedicated scripting language. As for security, the developer has to fill the application manifest file with the permissions required by the application, with Android being a privilege-separated operating system.

Limited resources The small form factor of smartphones brings computation and energy constraints, which restrict GP usage and require careful design of the GP infrastructure. Developers must carefully use AGP as it shouldn't contribute to undesirable user experiences, such as quick battery depletion. For instance, AGP can perform costly computation when energy resources are ample, and minimise learning when the user has high demand for the phone's resources or when the battery is running flat.

Battery level is easy to obtain through the Android API, which uses a coarse-grained scale from 0 to 100. However, real battery depletion can't be assessed from this battery level as the depletion rule differs from one device to another. We recommend to track a system file used by Android to record live consumption. This file is available on most of the devices.

To validate our framework, we demonstrate AGP with applications on two different generations of Android devices: the early HTC Magic running Android 1.6 ; and the more capable Nexus S, embedding a dual-core processor with the recent Android Gingerbread (version 2.3).

Services and Intent GP needs long-running background operations, which we implement using services under Android. An Android service is a component that can run in the background even when the user is not interacting with the device [16].

AGP implements one service for the Interpreter Shell which is used to process the programs generated. For each program launched, a new Interpreter Shell is created. By default, a service runs in the main thread of the application that hosts it. In our implementation, we forced the Interpreter Shell to be launched into a separate process in order to reduce Application Not Responding (ANR) errors when bugged programs are pushed.

Developers never directly deal with the Interpreter Shell. Rather, they launch it through a class called Interpreter. The latter attaches an Interpreter Context to the Interpreter Shell, which enables saving of variables and active links to other components during the program execution (cf. Figure 1).

Fig. 1. Interpreter package organization

The Interpreter Context is specifically needed for a GP framework running on a mobile OS such as AGP. Unlike GP frameworks running on desktop computers, we do not have direct access to some sensors. For example, an application may notify the Android layer that a sensor is no longer required, however the OS must consider others applications concurrently accessing the sensor before deciding whether to turn it off. Since a GP developer does not have full access to the system, they need to keep trace of the previous actions performed to provide the state of the sensors in the program context. The Interpreter Context provides this functionality.

Figure 2 provides a high level view of the AGP framework. The core AGP framework provides the infrastructure for writing new GP application for Android smartphones. An application using AGP will require the developer to implement at least two services which can easily communicate with a Communicator class.

3.3 Common Data Structures

Functions and Terminals Functions and terminals are the primitives of any GP system. Our framework allows reusability of functions and terminals for all applications. With AGP, the developer has to implement our specific Java interface for functions or terminals, respectively FunctionInterface and TerminalInterface. It specifies the required methods for the GP framework such as the primitive arity, the string representation used to serialize, the execution and the estimated time cost.

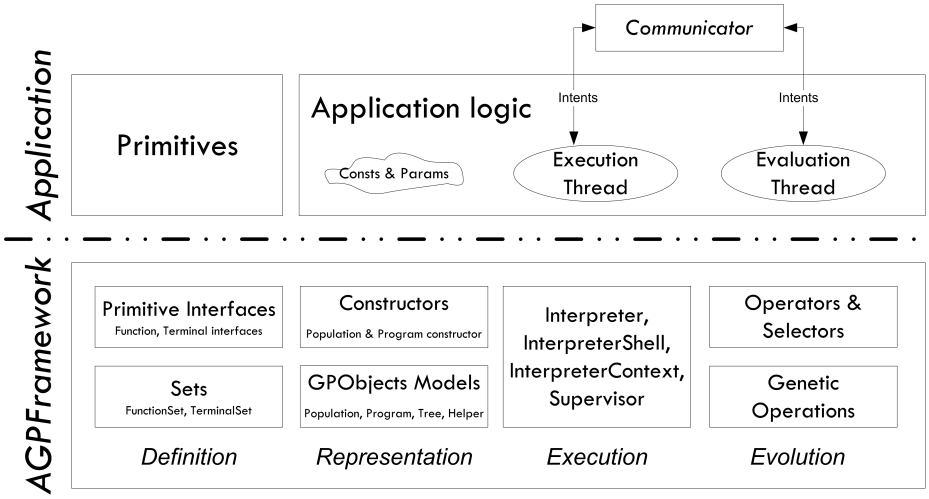


Fig. 2. Global Architecture of AGP

To provide flexibility for the developer to vary the selection of primitives for their application, we integrate the Strategy Pattern [21]. FunctionSet and TerminalSet classes respectively store the available functions and terminals for an application, those which respectively implement FunctionInterface and TerminalInterface. The developer can easily add, remove or look for a primitive through dedicated methods implemented in FunctionSet and TerminalSet.

Population and Programs Programs in AGP are represented as trees, and we define a population as a set of programs. We include the Builder Pattern [21] as a means for flexible population and programs inclusion into AGP. AGP already includes basic ways to get programs and populations, such as full random or empty set. Thanks to the Builder Pattern, the application developer is able to design their own program and population builders. For instance, to provide their program builder requires the implementation of the ProgramBuilderInterface (see Figure 3). It includes the getProgram() method which gets useful tools for creating programs such as the function and terminal sets, and a helper. We present the latter in Section 3.4.

Selectors and Genetic Operations When the evaluation thread is running alongside the execution thread, programs are evaluated and receive a fitness value. In GP, the programs that perform well are chosen to breed the next generation. Selectors are organized following the Strategy Pattern in order to provide flexibility. This way, the developer can easily switch between standard selectors provided by AGP such as the Wheel selector, or create his own selector solution.

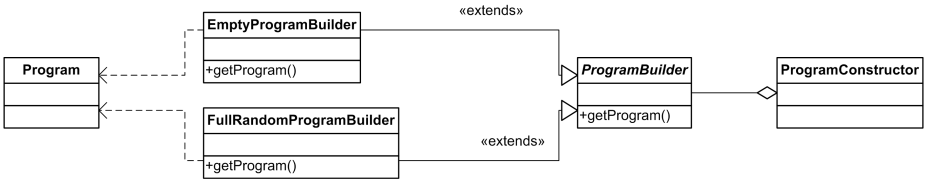


Fig. 3. Use of Builder Pattern in AGP illustrated with Program Constructor and two basics Program Builders. Abstract classes are shown in italic.

AGP then executes genetic operators on selected programs. In this regard, AGP currently supports two primary genetic operators widely used in GP, crossover and mutation, although it is extendible with more operators.

Save current states As smartphones are subject to reboot by the user, or undesired termination caused by battery depletion, AGP provides a safe mechanism to save the current state. We included serializable classes for the programs and the populations, which allow the developer to save these objects over reboot.

The developer is able to get program and population objects from their serialized form thanks to specific program and population builders (cf. Section 3.3), . Indeed, UnserializePopulationBuilder and UnserializeProgramBuilder are builders available in AGP, able to construct, respectively, populations and programs from serialized form saved into a file.

3.4 Improvements

As GP is a stochastic process, convergence towards desired performance can take several generations. Contrary to typical computers, which usually run GP algorithms, smartphones have limited energy and computational capabilities. In order to ensure that convergence time remains within reasonable limits, AGP includes two components, namely the helper and the supervisor, which enable the use of expert knowledge to apply constraints to AGP-generated programs.

Helper The helper is called during the program generation process done by any program builder. The developer can create one or several helpers for one application by implementing the AGP HelperInterface interface. Whenever a program is generated, AGP will refer to the helper evaluate() method to specifiy any correctness conditions that the program has to meet. For instance, in our geolocalization application (cf. Section 5), if a program doesn't call any location provider, we know that this program will be unable to locate the smartphone, so we can reasonably discard it without losing evaluation time.

Supervisor The supervisor runs during the program interpretation. It can check constraints on-the-fly, and kill the Interpreter Shell if the program goes out of

bounds. For instance, the supervisor can control execution time. If the program execution is too long, the supervisor will automatically kill the program.

4 Google Reader application

4.1 Application purpose

Google Reader is a web-based aggregator released and maintained by Google. It works as a RSS feed reader, allowing users to get latest news from selected feeds. Many applications exist for consulting feeds from smartphones. However, news you like to read on a smartphone may depend on the kind of content, and its readability on a mobile device. For instance, it is easy to read short text news whereas it is uncomfortable to look for long articles, comics, infographics or flash animations. Moreover, such content might quickly deplete the user monthly capped data plan. Our Google Reader GP (GRGP) application takes advantage of the AGP framework to learn which feeds the user likes to read on their smartphone. Obviously, the feeds will be taken from the user Google Reader account.

The application is kept simple for demonstration purposes: whenever the user wants to get news, she asks for a news report which executes a GP program and returns the latest and unread news from feeds selected by the program.

4.2 Fitness definition

The fitness definition is based on two sub-fitness functions:

$$Fitness = SubFitness_{News\ \ count} \times SubFitness_{News\ \ clicked}$$

$$SubFitness_{News\ \ count} = \begin{cases} \frac{Displayed\ \ news}{Desired\ \ quantity} & Disp.\ \ news \leq Desired\ \ qty. \\ 1 & otherwise \end{cases}$$

$$SubFitness_{News\ \ clicked} = \left(\frac{Clicked\ \ news}{Displayed\ \ news} \right)$$

The news count refers to the minimum desired quantity of displayed news whenever the user requests a news report. This is a setting fixed by the user in the configuration menu. The clicked news corresponds to the quantity of news read over the amount of news given in the report. In other terms, it evaluates the interest of the user in the displayed news.

4.3 Results

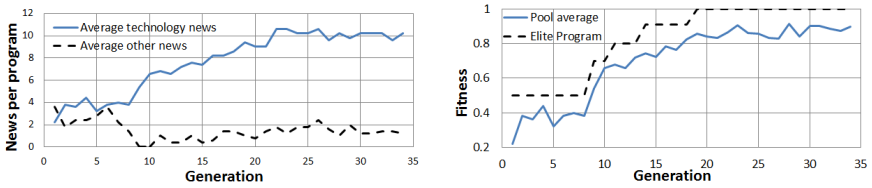
We conducted our experiment with 7 sources from a real Google Reader subscription: 4 technology news websites (TechCrunch, TechLand, Engadget and Digital Trends), VisualLoop which gives fresh infographics, Break Videos for

funny videos, and Business Green for latest green products. Even though the concerned user was interested in all those sources, he is used to read the technology news on his smartphone rather than the other sources providing longer articles or heavy media files not optimised for reading on smartphone.

Figure 4(a) reports the average of displayed news per program over generations. In order to provide a clean graph, we grouped the news feed in two sub-groups: the 4 technology news, and the others (VisualLoop, BreakVideos, and Business Green). After 6 generations, our Google Reader using AGP learned the user preference for the technology news. However, it doesn't eliminate completely the diversity and keeps proposing some news from the other feeds yet with a lower likelihood.

As expected, GRGP learns to provide the desired minimum quantity of news per report, which we set to 10 for this experiment. Figure 4(a) confirms this convergence by looking at the sum of technology and other news.

The program fitness evolution indicates that our evolution strategy does its job, and leads to an increase of the elite program fitness. In Figure 4(b), the pool average represents the mean pool fitness (i.e. mean fitness of the 5 programs). Whenever the elite program gains in fitness, it subsequently leads to an increase of the pool average fitness.



(a) News Distribution (b) Program fitness evolution

Fig. 4. News Distribution and Program fitness evolution

5 Context-aware localization

With their application programming interface, modern smartphones OS enable programmers to develop their own solutions using available sensors on the device to get user context [17,18]. However, frequent usage of sensors remains a problem as it quickly depletes batteries. Thus, a key challenge is ensuring sensor sampling provides sufficient context without affecting battery lifetime. While user motion, sound activity or ambient light can be retrieved from one sensor, getting the position is a more complex problem as it can be obtained from several sources: ranging from the energy-hungry yet accurate GPS to the energy-efficient yet inaccurate cell-based method that relies on cellular phone towers. The various localization options on smartphones requires consideration of their availability and their energy/accuracy trade-off [20].

As this problem depends on too many contextual constraints as position, signal quality, device energy profile, it is unlikely to foretell which algorithm is going to fit better than others for a user in a specific environment. The dynamic changes in constraints motivate the use of methods that can not only adapt to expected changes but can also learn how to deal unexpected changes in context. We introduce an application using AGP aimed to address this problem. We focus here on AGP's ability to achieve results for a problem depending on many contextual constraints.

5.1 Fitness function definition

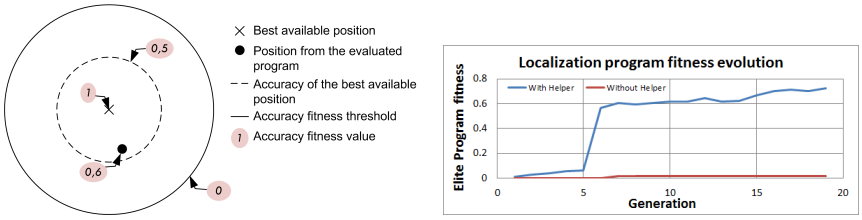
The fitness function used in our localization application reflects the common trade-off between energy and accuracy in the localization field [19]. We introduce two fitness metrics: the accuracy fitness and the energy fitness, which respectively quantify the accuracy and energy efficiency of the provided solution. As positions are dynamic, we evaluate fitness every second during the evaluation period (n seconds). By the end of the evaluation time, we use the average of these subfitnesses to give a fitness to the program. The overall fitness is obtained by multiplying these subfitnesses. We choose this option as it discards any solution which doesn't provide any accuracy or could deplete the battery, while maintaining simplicity for the demonstration purpose of this paper.

$$Fitness = \frac{\sum_{i=1}^{n} SubFitness_{Accuracy}(i) \times SubFitness_{Energy}(i)}{n} \quad (1)$$

Accuracy fitness In Android, localization can be achieved through several location providers used alone or combined. Usual location providers are GPS, Cellular Network and Wi-Fi. The developer could also brings other providers such as a contact-logging beacon method [20], or an accelerometer assisted algorithm. When the application is learning, the evaluation process keeps all the location provider on. It automatically picks the provider giving the most favorable accuracy. We call the output from this provider the best available position.

We use this best available position as a reference to attribute an accuracy fitness to the position provided by the evaluated program (cf. Figure 5(a)):
- if program position is within the accuracy of the best available position, we attribute an accuracy fitness following a linear rule from 1 to 0.5
- if program position is outside the former, but within a circle of twice the accuracy of the best available position (accuracy fitness threshold), we attribute an accuracy fitness following a linear rule from 0.5 to 0.

Energy fitness We define energy fitness according to a basic rule: we want to provide localization without depleting the battery by the end of the day as we assume users can charge their phones at the end of each day. We consider an average 1400 mAh battery capacity for the paper. To achieve our goal, the



(a) Fitness accuracy (b) Fitness evolution

Fig. 5. The localisation application with AGP

average power consumption should not exceed 63 mA (= 1400 / 22 hours). We define a day as 22 hours because we consider the phone as plugged for 2 hours per day.

We use the Android PowerProfile class to estimate the power consumption per chip. We access this class through the Java reflection mechanism. This enables AGP to assess the power cost of the evaluated program depending on the CPU usage and the chip used to locate the smartphone. For simplicity, we limit the energy fitness to a linear function, ranging from 1 for a idealistic case where the program doesn't cost any power to 0 for a program which requires more power than the one day energy budget.

5.2 Results

We conduct the experiment with populations of 12 programs. The program evaluation is limited to one minute. Our function set contains general operators such as addition, multiplication, and other application-specific operators: functions to switch location providers such as GPS, Wi-Fi or Cellular Network. These functions need access to some Android components, which is possible with AGP's design. Figure 5(b) shows the framework ability to get a program localizing the smartphone. After a rough first solution, it converges to smarter programs able to provide more efficient and accurate solutions.

We also conduct an experiment to evaluate the benefits from the use of Helper. Populations generated with the Helper provide a working solution in the first generation, and quickly have satisfying programs. On the other hand, populations generated without the Helper are stuck with non-working programs (zero fitness) for several generations. Then, they only get a slow evolution. It is mainly due to many programs which make no sense: they don't switch on a location provider or don't call any latitude nor longitude update.

6 Discussion and Conclusion

The innate inter-communication capability between mobile devices lends itself to the Island Model implementation where each mobile device hosts a population

and evolved programs can be serialised and shared (migrated) [22]. While intuitively one expects the parallel resources will expedite convergence, the Island model is also known to generate better quality solutions [23]. As the framework currently does not implement such cooperative evolution mechanisms, our evolution is constrained by the modest computational resources of the mobile device. As such, we have attempted only simple problems where a single small population can converge within a reasonably short period. This configuration however would likely require very long convergence times for more complex problems.

The symbolic nature of GP means that logic can be readily seeded and is an intuitive choice where control of device's resources is typically performed programatically. It should be noted, however, that adaptive behaviour may not be desirable for all interactions. For example, the user interface should have a consistent feel across applications and adhere to the device or OS recommended UI design recommendations. However, within this constraint, adaptive behaviour may provide a method to overcome consistently undesirable application behaviour. The current configuration employs a fixed population structure which produces a number of likely low performing programs to be evaluated every generation even after the system has converged. This means that there will always be some 'annoying' behaviour. Ideally the population generation operations over time would change to converge the population to only desirable behaviour, however this would also reduce the ability of the system to respond quickly to changes in user preferences.

This paper has presented Android GP Framework (AGP) as the first genetic programming framework for the mobile Android OS. The framework considers the resource constraints and programming restrictions on Android smart phones for evolving logic, and introduces special components for quicker convergence by limiting execution to meaningful programs. We demonstrated AGP's capabilities with two applications and showed that it can converge to desirable performance quickly towards objective functions with multiple constraints. We believe AGP represents a first step towards versatile online personalisation in the growing smart phone market.

Acknowledgements

The authors would like to thank Brano Kusy for his valuable inputs in realising this work. This project was supported by the Sensors and Sensor Network Transformational Capability Platform at CSIRO.

References

1. Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., Estrin, D.: Diversity in Smartphone Usage. In: MobiSys'10. (2010)
2. Dockhorn Costa, P., Ferreira Pires, L., Sinderen, M., 2008. Designing a con?gurable services platform for mobile context-aware applications. International Journal of Pervasive Computing and Communications 1 (1), 1325.

3. Bae, J.S., Lee, J.Y., Kim, B.C., Rye, S., 2006. Next Generation Mobile Service Environment and Evolution of Context Aware Services, International Conference, EUC 2006, Seoul, Korea, August 1-4.
4. Miele, A., Quintarelli, E., Tanca, L.: A methodology for preference-based personalization of contextual data. In: EDBT 2009. (2009)
5. Koza, J.R.: Genetic Programming : on the programming of computers by means of natural selection. In: Complex adaptive systems, MIT Press, Cambridge. (1992)
6. Ventura, S., Romero, C., Zafra, A., Delgado, J. A., Hervás, C.: JCLEC: a Java framework for evolutionary computation. In: Soft Comput., vol. 12, pp. 381-392. (2008)
7. Pyevolve, `http://pyevolve.sourceforge.net/wordpress/?p=350`
8. McPhee, N. F., Hopper, N. J., Reierson, M. L.: Sutherland: An extensible object-oriented software framework for evolutionary computation. In: Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin. San Francisco, CA: Morgan Kaufmann. (1998)
9. Gagn, C., Parizeau, M.: Open BEAGLE: A New Versatile C++ Framework for Evolutionary Computation.In: GECCO Late Breaking Papers, pp. 161-168. (2002)
10. Ismail, I.A., Ramly, N.A.E., Abd-ElWahid, M.A., ElKafrawy, P.M., Nasef, M.M.: Genetic Programming Framework for Fingerprint Matching.In: CoRR. (2009)
11. Torres, R. S., Falco, A. X., Gonalves, M. A., Papa, J. P., Zhang, B., Fan, W., Fox, E. A.: A genetic programming framework for content-based image retrieval. In: Pattern Recognition, vol. 42, pp. 283 - 292. Elsevier (2009)
12. Valencia, P., Lindsay, P., Jurdak, R.: Distributed Genetic Evolution in WSN. In: IPSN'10, April 12-16, 2010, Stockholm, Sweden. (2010)
13. Lenaerts, T., Manderick, B.: Building a Genetic Programming Framework. The Added-Value of Design Patterns. In: Proceeding EuroGP'98, 1998. (1998)
14. Lacerda, A., Cristo, M., Gonalves, M.A., Fan, W., Ziviani, N., Ribeiro-Neto, B.A.: Learning to advertise.In: SIGIR'06, pp. 549-556. (2006)
15. Weise, T., Geihs, K.: DGPF: An Adaptable Framework for Distributed Multi-Objective Search Algorithms Applied to the Genetic Programming of Sensor Networks. In: BIOMA'06, October 910, 2006, pages 157166, Ljubljana, Slovenia. (2006)
16. Android Reference, `http://developer.android.com/reference/packages.html`
17. Lu, H., Pan, W., Lane, N.D., Choudhury, T., Campbell, A.T.: SoundSense: scalable sound sensing for people-centric applications on mobile phones.In: MobiSys, pp. 165-178. (2009)
18. Thiagarajan, A., Ravindranath, L., LaCurts, K., Madden, S., Balakrishnan, H., Toledo, S., Eriksson, J.: VTrack: accurate, energy-aware road traffic delay estimation using mobile phones.In: SenSys, pp. 85-98. (2009)
19. Lin, K., Kansal, A., Lymberopoulos, D., Zhao, F.: Energy-accuracy trade-off for continuous mobile device location.In: MobiSys, pp. 285-298. (2010)
20. Jurdak, R., Corke, P., Dharman, D., Salagnac, G.: Adaptive GPS duty cycling and radio ranging for energy-efficient localization.In: SenSys, pp. 57-70. (2010)
21. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-oriented modelling and design. (1991)
22. Folino G., Spezzano G., P-CAGE: An Environment for Evolutionary Computation in Peer-to-Peer Systems, EuroGP 2006, LNCS vol. 3905, pp. 341-350, Springer Verlag (2006)
23. W. N. Martin, J. Lienig and J. P. Cohoon: Island (Migration) Models: Evolutionary Algorithms Based on Punctuated Equilibria, in Handbook of Evolutionary Computation, pp. C6.3:1-C6.3:16, Oxford University Press, (1997)