The Effects of Developer Dynamics on Fitness in an Evolutionary Ecosystem Model of the App Store

Soo Ling Lim, Peter J. Bentley, and Fuyuki Ishikawa

***Abstract*—Natural ecosystems exhibit complex dynamics of interacting species. Man-made ecosystems exhibit similar dynamics and, in the case of mobile app stores, can be said to perform optimization as developers seek to maximize app downloads. This work aims to understand stability and instability within app store dynamics and how it affects fitness. The investigation is carried out with AppEco, a model of the iOS App Store, which was extended for this paper and updated to model the store from 2008 to 2014. AppEco models apps containing features, developers who build the apps, users who download apps according to their preferences, and an app store that presents apps to the users. It also models developers who use commonly observed strategies to build their apps: innovator, milker, optimizer, copycat, and flexible (the ability to choose any strategy). Results show that despite the success of the copycat strategy, there is a clear stable state for low proportion of copycats in developer populations, mirroring results in theoretical biology for producer–scrounger systems. The results also show that the best fitness is achieved when the evolutionary optimizer (as producer) and copycat (as scrounger) strategies coexist together in stable proportions.**

***Index Terms*—Agent-based simulation, app stores, computational modeling, evolutionary ecosystem model, genetic algorithms, mobile app developers, producer–scrounger systems.**

I. Introduction

THE interdependencies of populations within ecosystems are complex. They may be competitive or cooperative; species may behave as parasite, symbiont, prey, or predator. Any measure of ecosystem health may be arguable—is it equilibrium? Stability of populations? Longevity? But separate from any notion of health, the dynamics of a sustainable ecosystem[1] are of great interest. Such dynamics, if harnessed within evolutionary computation for the purposes of search or optimization, could have great potential. If the dynamics of crossover and mutation can drive search within a genetic algorithm, and the dynamics of agent interactions within a swarm can drive search within a particle swarm optimization algorithm, then can population dynamics within an ecosystem drive search in a new and useful way? If so, what kinds of dynamics exist that might be useful?

To address these questions, and because of the availability of data describing its behavior, in this paper we focus on a man-made evolutionary ecosystem: the commercial software environment known as the app store. We investigate app store dynamics, focusing on the stability and instability of developer strategies and how they affect fitness.

App stores provide the environment for hundreds of thousands of software applications (apps) for mobile devices. Within this environment, apps compete with each other to receive downloads from users. Apps with well-implemented features that match user preferences are likely to become more successful.[2] Those with poorly implemented features or features that are not desirable will not receive downloads and so will be unsuccessful. Apps have a location in their app store environment, which may depend on their success. More successful apps are able to enjoy the rich pastures of the top apps or new apps charts, where they are considerably more likely to receive more downloads. Less successful apps find themselves relegated to obscurity, unnoticed by users.

Unlike living systems, apps do not reproduce themselves or develop from zygote to adult organism. However, apps are developed. Human developers follow a range of evolutionary strategies as they create new apps. Common strategies include releasing new apps based on modifications of their better performing apps, copying successful apps developed by others, and adding features from other successful apps to their own apps. Developers also create the strategies of interaction and competition between apps: they might attract attention to apps through advertising or buying positive feedback, they might encourage spread through the user population by making their

Manuscript received May 12, 2015; revised August 14, 2015; accepted October 14, 2015. Date of publication October 26, 2015; date of current version July 26, 2016.

S. L. Lim is with the Department of Computer Science, University College London, London WC1E 6BT, U.K., and also with the School of Design, Engineering and Computing, Bournemouth University, Poole BH12 5BB, U.K. (e-mail: s.lim@cs.ucl.ac.uk).

P. J. Bentley is with the Department of Computer Science, University College London, London WC1E 6BT, U.K. (e-mail: p.bentley@cs.ucl.ac.uk).

F. Ishikawa is with the National Institute of Informatics, Tokyo 101-8430, Japan (e-mail: f-ishikawa@nii.ac.jp).

This paper has supplementary downloadable multimedia material available at http://ieeexplore.ieee.org provided by the authors.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2015.2494382

[1]For the purposes of this work we define a sustainable ecosystem to be an evolutionary system in which at least two species coexist with nonzero population sizes, their fitnesses are interdependent, and their continued existence does not depend on outside influence.

[2]Where "features" include every aspect of the app, e.g., functionality, appearance, educational value, entertainment value, and so on.

apps "viral" or they may even attempt to kill competitor apps through leaving negative feedback.

The result is a highly competitive evolutionary ecosystem where apps and their developers compete in complex interactions that change constantly. In the app store, there is a constant pressure toward the creation of apps that can receive more downloads than their competitors, in whatever niche or category the apps belong. In this sense, the app store as a whole optimizes apps to meet user preferences.[3]

For this paper, we use AppEco [1]–[4], an agent-based evolutionary ecosystem model of the app store, as a test bed in order to perform an analysis of potentially useful evolutionary dynamics exhibited by an ecosystem. Previous work investigated AppEco by simulating the iOS App Store from 2008 to 2011 and investigating the performances of developer strategies [3], the effect of publicity on app downloads [1], and app ranking algorithms [4]. To ensure that these dynamics resemble those exhibited by the real app store as much as is feasible, for this work we use data gathered from a survey of 10 000 people worldwide [5] and incorporate new features into AppEco such as app feature quality, developer capabilities, and app ratings. We calibrated AppEco to simulate the iOS App Store from 2008 to 2014.

We focused on the causes of stability and instability in developer strategies and investigate their effects on app fitness. We considered the following research questions.

1) Is there an equilibrium for developer strategies in the app store? For example, is there cyclic behavior between "predator" and "prey" behaviors, and how does it influence fitness?
2) What are the attractors in the strategy state space of the app store? For example, are some strategies always more successful than others, and do they help improve fitness?
3) What is the relationship between stability and fitness?

The rest of the paper is organized as follows. Section II describes existing work. Section III describes AppEco. Section IV describes our calibration of the AppEco model to simulate the iOS app ecosystem, and Section V describes the baseline AppEco behavior. Section VI describes the experiment to answer the first research question, Section VII describes the experiment to answer the second research question, and Section VIII describes the experiment to answer the third research question. Section IX provides the conclusion and discusses future work.

II. Background

Evolutionary computation techniques have been used extensively in ecosystem modeling, as they exhibit similar properties to biological systems (e.g., genotype/phenotype mapping, adaptation, and evolution) and they can handle large search spaces efficiently [6], [7]. Existing models of nature-inspired evolutionary ecosystems include Daisyworld [8], Echo [9], Tierra [10], Swarm [11], and SugarScape [12].

There has also been increasing research that uses evolutionary agent-based modeling to explore and understand human systems such as financial markets [13]–[17], organizational behavior [18]–[21], foreign exchange markets [22], labor markets [23], [24], electricity markets [25], [26], and theoretical studies of agent interaction [27]–[31].

Existing models that are particularly relevant to this research have elements of evolutionary learning, and competition. For example, an agent-based model was developed to analyze the evolution of output decisions in a market that is dominated by a small number of firms [32]. In every time period, each firm decides whether to produce the same product as before, introduce a new product variant for an initial advantage, or copy a product produced by another firm. Different firms have different abilities to develop products and imitate existing designs; therefore, the effects of the decision whether to imitate or innovate differ between firms. The firms have simple rules to estimate market potentials and market founding potentials of all firms and make their decisions using a stochastic learning rule. Experiments were conducted to explore how the innovation rule of a firm should adapt for optimal profitability, both to the individual firms and to the industry as a whole.

More recently, an experimental simulation of cultural evolution was conducted to investigate the "copy-successful-individuals" learning strategy [33]. The experiment used a virtual arrowhead task where participants designed virtual arrowheads and tested them in virtual hunting environments, improving their designs through either individual trial-and-error learning or by copying the design of another participant. Results showed that the copy-successful-individuals strategy was significantly more adaptive than individual learning. The adaptive advantage of copy-successful-individuals was maintained when cultural learning was permitted at regular intervals.

This background barely scratches the surface of the literature on evolutionary ecosystem models (for more extensive reviews see [34], [35]), but it illustrates the rich diversity of modeling approaches and the strong benefits of creating such models as an aid to understanding the complex dynamics of natural and human systems. In this paper, we build on these previous studies and investigate one of the newest forms of human ecosystem, using AppEco, a simulation of the app store.

III. AppEco

AppEco is an evolutionary model of the mobile app store ecosystem, which uses a genetic algorithm as inspiration for its representation. It models, apps as grids of features (akin to individuals with genes). It models developers, users, and the app store environment. The model is informed by analyzing mobile app stores, conducting a survey of more than 10 000 people from over 15 countries about their mobile app usage behavior [5], and interviews of app developers.

AppEco models an ever-growing population of apps held in an app store. Developer agents build and upload apps to

[3]Not all developers wish their apps to be the best in the store. However, the majority of developers wish their apps to be downloaded by users who desire or require their app, and hence their apps must compete for downloads with all other apps that have the same features. Apps with better features or better-implemented features, which better meet the user preferences, will receive more downloads.

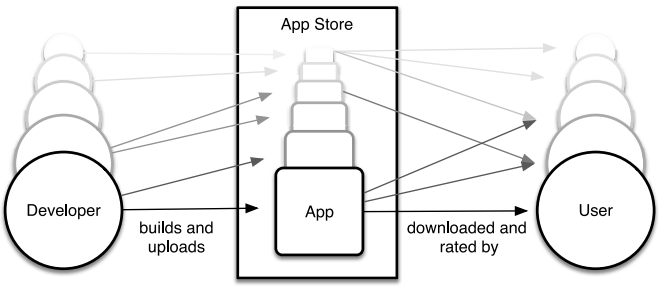

Fig. 1. Interactions between agents via apps in AppEco adapted from [1].

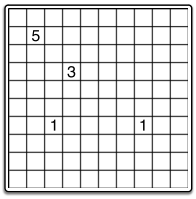

Fig. 2. App feature grid **F** (empty cells are zeroes, not shown for clarity).

the app store using principles of inheritance and variation; user agents browse the app store and select apps to download, see Fig. 1. Apps are more successful if they receive more downloads.

A. AppEco Components

AppEco consists of four key components: 1) apps; 2) developers; 3) users; and 4) the app store. We describe each component below.

1) Apps: Inspired by the way genetic algorithms encode sets of parameters as chromosomes, each app is modeled as a 10×10 feature grid (**F**). Each cell in **F** has a probability P_{Feat} of being nonzero. If a cell in **F** is nonzero, then the app offers that particular feature. Each nonzero cell has a quality attribute, which is a number between 1 and 5, where a higher value corresponds to a better quality implementation of that feature (Fig. 2).

Each app records its total downloads and downloads for each of the last seven days. Apps also have an overall rating between 1 and 5 stars, which is an average of all the "star ratings" provided by users. Each app also records the time at which it was uploaded.

Apps may be infectious with probability $P_{Infectious}$. Infectious apps are recommended across the social network of users, each user informing its friends to browse the app. This models apps that might have addictive or competitive features, or apps that enable communication between users thus requiring all users to have the app. Once created, most apps remain for sale on the app store.[4]

2) Developers: Unlike living organisms, apps do not reproduce directly. Instead, they are created by developers who use evolutionary strategies to make new apps, inheriting features from other apps. Evolutionary selection of apps is caused by the selection of apps to download by users, according to their user preferences. Each developer uses one of the following five strategies to build apps. These strategies are derived from app literature and consultation with app developers.

1) *S0 Innovator:* Apps are built with random features. This simulates innovative developers who develop diverse and random apps. For example, iOS developer Shape Services produced apps in a variety of different categories such as social networking, utilities, and productivity.[5] To implement this, the cells in **F** are filled probabilistically, such that each cell in the grid has a probability P_{Feat} of being nonzero.
2) *S1 Milker:* A variation of the developer's own most recent app is built each time. This simulates developers who "milk" a single app idea repeatedly. An example is Brighthouse Labs, which produced thousands of similar apps,[6] such as an app to provide regional news for each area in each country, and an app for each sports team of multiple sports. To implement this, the developer copies the features of his own latest app with random creep mutation. (For the first app the cells in **F** are filled probabilistically as in S0.)
3) *S2 Optimizer:* A variation of the developer's own best app is built each time. This simulates developers who improve on their best app. For example, Rovio developed many game apps before finding success with Angry Birds. They then released new apps such as Angry Birds Seasons, and Angry Birds Rio.[7] To implement this, the developer copies the features in his own best app (the app with the highest daily average downloads) with random mutation. If no apps by this developer have downloads, the developer just copies his most recent app. (For the first app the cells in **F** are filled probabilistically as in S0.)
4) *S3 Copycat:* An app in the top apps chart is copied. This simulates developers who are less creative but desire many downloads. Angry Chickens and Angry Dogs are example copycats of Angry Birds.[8] To implement this, an app is randomly selected from the top apps chart and its features are copied with random mutation.
5) *S* Flexible:* The first app is developed with one of the strategies S0–S3 (randomly chosen). After submitting their first app, each developer will randomly select an app from the top apps chart and change strategy to be the same as the developer of the selected app (with probability 0.99). This models developers learning about successful developer strategies via blogs and articles in the media. A random strategy is selected with probability 0.01.

From the perspective of evolutionary computation literature, each strategy behaves like a different search algorithm running in parallel, and each exploits a different form of learning (Table I).

[4]Apps can be removed by the app store if they break app store rules or are no longer supported by the app store due to software updates. They can also be removed by developers. However, these are a very small number of apps.

[5]http://www.shapeservices.com/

[6]http://isource.com/2009/05/27/app-store-hall-of-shame-brighthouse-labs/

[7]http://www.wired.co.uk/magazine/archive/2011/04/features/how-rovio-made-angry-birds-a-winner

[8]http://techcrunch.com/2011/12/06/can-we-stop-the-copycat-apps/

TABLE I
DEVELOPER STRATEGIES AND THEIR EQUIVALENT SEARCH ALGORITHMS AND LEARNING STRATEGIES

Developer Strategy	Equivalent Search Algorithm	Evolutionary Learning Strategy
S0 Innovator	Random search.[10]	No learning.
S1 Milker	Random walk.	No learning.
S2 Optimizer	Simple (μ +1) EA, where μ is the number of apps by that developer.	Individual learning.
S3 Copycat	No direct EA analogy; shares some properties with elitist GAs.	Cultural learning of successful apps.
S* Flexible	Ensemble learning, using the previous four algorithms.	Cultural learning of successful strategies.

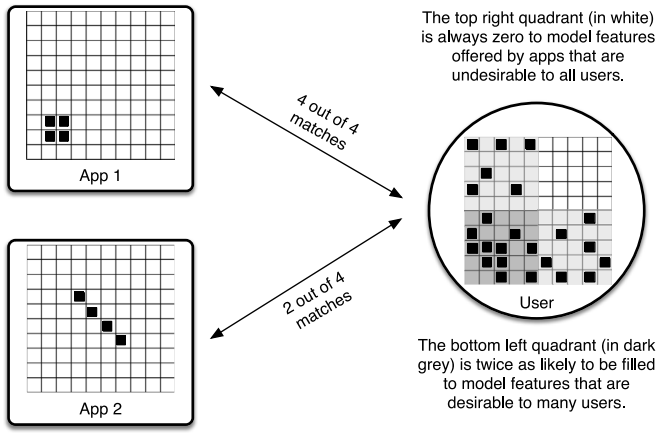

Fig. 3. Matching user preferences with app features [1]. Each dark cell in the user grid is filled with a random number between 1 and 5. Each dark cell in the app grid is filled with a random number between 1 and the developer's capability.

The app feature grid **F** is filled depending on the strategy used by the app's developer. Every developer has a capability level, which determines the maximum quality of features within the apps it develops. Every nonzero feature added to an app is randomly chosen between 1 and the maximum capability of that developer. For most of the strategies, app features are copied from existing apps to the new app. If the app that is being copied belongs to the same developer, the quality of one of the features will change (mutate) during the copy. This is to simulate developers improving the quality of their app or accidentally making it worse by reworking the app. If the app belongs to a different developer, each feature of the app has the quality of between 1 and the maximum capability of the developer who is doing the copy, i.e., the developer copies the features but not the quality. One of the following mutations occurs during each copy.

1) *A feature is removed:* A nonzero cell is randomly selected and removed in **F**.
2) *A feature is added:* An empty cell is randomly selected and filled in **F**.
3) *A feature is moved:* A nonzero cell is randomly selected and moved to a randomly selected empty cell in **F**.

Each developer agent models a solo developer or developer team. Each developer agent takes *devDuration* (a random value between [dev_{min}, dev_{max}]) days to build an app and uses *daysTaken* to record the number of days spent building the app so far. Developers are initially active (they continuously build and upload apps to the app store). They may become inactive with probability $P_{Inactive}$ to model part-time developers, hobbyists, and the tendency of developers to stop building apps.[9] Developers also record the number of apps they have developed and the downloads received.

3) Users: Each user agent uses a 10 × 10 preference grid (**P**) to specify which app features it desires. Universally undesirable features are modeled by setting the top right quadrant of **P** to zero. For example, no users want an app that is difficult to use or malicious. Features that are desirable to some users are modeled by filling, with probability P_{Pref}, the top left and bottom right quadrant in **P** with a random number between 1 and 5. This simulates the intensity of the user preference where 1 is low and 5 is high. Finally, more desirable features are modeled by filling, with probability $2 \times P_{Pref}$, the bottom left quadrant in **P** with a random number between 1 and 5. The right half of Fig. 3 illustrates an example preference grid. User preferences are unknown to developers.

Filled (nonzero) cells in **P** represent features of **F** that are desired by the user. If an app's feature grid **F** has a corresponding cell filled, then the app offers the feature desired by the user agent. For example, in Fig. 3, all of the four features of App 1 match the preferences of the user agent, but only two of the features of App 2 match the user agent's preferences. A highly desirable app can be abstracted as an app with **F** that matches **P** of many users, while a highly undesirable app has **F** that matches **P** of few or no users. Preference matching is binary: filled cells either match or do not match. Users cannot assess the quality of features until they have downloaded and tried the app.

Each user has a probability $P_{ProvideRatings}$ of providing ratings. Users who provide ratings do so the form of a "five-star" value after they have downloaded an app. Users rate an app by first calculating *featRating* for each feature of the app following Table II. For example, for a given feature, if the feature has a quality of 2 and the user has a preference intensity for the feature of 4, then *featRating* is 3. These ratings are then averaged to produce the overall *rating* for the app

$$rating = \frac{\sum_{i=0}^{numFeat} featRating_i}{numFeat} \quad (1)$$

where *numFeat* is the total number of features of the app and *featRating* is the rating for each feature of the app.

Each user has a probability $P_{InfluencedByRatings}$ of being influenced by an app's overall rating when they download an app. An app's overall rating is then the total ratings divided by the number of ratings it has received. The distribution of overall app ratings in AppEco resembles the actual iOS App Store

[9] http://t-machine.org/index.php/2009/06/11/may-2009-survey-of-iphone-developers/

[10] There is heuristic value in imagining that innovation, like mutation, is essentially random [36].

TABLE II
LOOKUP TABLE FOR FEATURE RATING [FOR EACH POSSIBLE FEATURE QUALITY (1–5), THE TABLE PROVIDES THE FEATURE RATING (1–5) FOR EACH POSSIBLE PREFERENCE INTENSITY (1–5)]

Preference intensity \ Feature quality	1	2	3	4	5
1	5	5	5	5	5
2	4	5	5	5	5
3	3	4	5	5	5
4	2	3	4	5	5
5	1	2	3	4	5

where 4-star is the most common rating.[11] If a user is influenced by ratings, the user avoids apps with an average rating of less than 3.

Each user agent records the apps it downloaded, the period of time between each browse of the app store (*daysBtwBrowse*, a random value between [bro_{min}, bro_{max}]), and the time elapsed since it last browsed the app store (*daysElapsed*), so that it knows when to browse again. *daysElapsed* is initialized to a random number between [0, *daysBtwBrowse*] so that users do not browse at the same time when they start.

Each user may influence a number of friends to examine an app that the user has just downloaded, *numFriends*. *numFriends* is a random number with a power law distribution in the range [0, 150]. This simulates the finding that many people influence very few friends, and a few people influence many friends. The upper limit of this range is derived from the Dunbar number of 150 [37].

4) App Store: The app store hosts apps created by developers and makes them available for browsing and downloading by users. It provides three browsing methods: 1) the top apps chart; 2) the new apps chart; and 3) keyword search. These browsing approaches reveal different subsets of apps to users as follows.

The top apps chart provides an ordered list of the top $N_{MaxTopChart}$ apps with highest downloads (default 50). The new apps chart provides a random sampling ($P_{OnNewChart} = 0.001$) of $N_{MaxNewChart}$ newly uploaded apps (default 40). Keyword search returns a list of apps that match the search term entered by the user. In AppEco, keyword search is abstracted as a search which returns a random number of randomly chosen apps between [key_{min}, key_{max}].

B. AppEco Algorithm

The AppEco algorithm models the daily interactions between the AppEco components described in the previous section. The population growth of user agents is modeled using a sigmoid growth function as used for population growth in natural systems [38]. The equation models a declining growth rate of user agents in an app ecosystem as the population density increases, representing the market share limit of the mobile platform.[12] The user population size at timestep t, $userPop_t$, is defined as

$$userPop_t = 1000 + \frac{79000}{1 + e^{0.0028t+4}} \tag{2}$$

where the values are calibrated to match the real world data as described in Section IV.

The population of developers is modeled using a polynomial function derived from the real world data. The developer population size at timestep t, $devPop_t$, is defined as follows:

$$devPop_t = 0.045t^2 + 40t + 1200 \tag{3}$$

where the values are calibrated to match the real world data as described in Section IV.

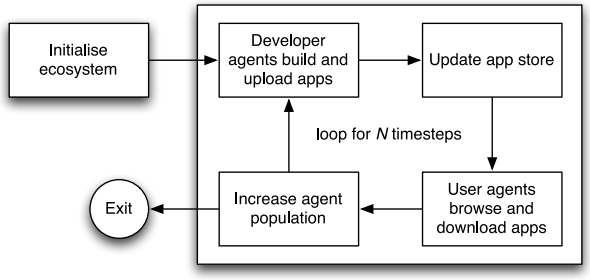

Fig. 4. AppEco algorithm [1].

Fig. 4 summarizes the AppEco algorithm. Every timestep in the AppEco algorithm models one day of interactions between developers, users, apps, and the app store. The following sections describe each major component of the algorithm.

1) Initialize Ecosystem: AppEco is initialized at timestep $t = 0$ with populations of user and developer agents calculated using (2) and (3). $N_{InitApp}$ initial apps are also created by randomly selected developers. (The iOS App Store was launched with 500 apps.)[13] All attributes of the initial developers, apps, and users are set as described earlier.

2) Developer Agents Build and Upload Apps: For each developer, if the developer is active, *daysTaken* is incremented. If *daysTaken* exceeds *devDuration* then the feature grid **F** of a new app is set according to the developer strategy of the developer. The app is uploaded to the store and *daysTaken* is reset to 0.

3) Update App Store: The new apps chart is updated. At timestep $t = 0$, the new apps chart comprises a random selection of $N_{MaxNewChart}$ initial apps. For subsequent timesteps, each new app will appear on the new apps chart with probability $P_{OnNewChart}$. The chart is implemented as a queue with older apps pushed out of the charts when their position exceeds the chart size. The top apps chart is also updated. At timestep $t = 0$, there are no downloads so the chart is empty. For subsequent timesteps, apps are sorted in descending order of their *score*, calculated using

$$score = 8D_1 + 5D_2 + 5D_3 + 3D_4 \tag{4}$$

[11] http://blog.scottlogic.com/2014/03/20/app-store-analysis.html

[12] The model does not simulate a future negative rate of growth as this is not evident in the iOS App Store data for the period of time we simulate. In future years, it is likely that technology will change and the app stores will decline, resulting in a contraction of all populations.

[13] http://www.apple.com/pr/library/2008/07/10iPhone-3G-on-Sale-Tomorrow.html

TABLE III
CONSTANTS AND VALUES FROM iOS CALIBRATION

Constant	Values
$N_{InitApp}$	500
$[bro_{min}, bro_{max}]$	[1, 360]
$[dev_{min}, dev_{max}]$	[1, 180]
P_{Pref}	0.35
P_{Feat}	0.04
$P_{OnNewChart}$	0.001
$N_{MaxNewChart}$	40
$N_{MaxTopChart}$	50
$P_{Inactive}$	0.0048
$[key_{min}, key_{max}]$	[0, 50]
$P_{ProvidRatings}$	0.4737
$P_{InfluencedByRatings}$	0.4811
$P_{Infectious}$	0.0001

where D_n is the number of downloads received by the app on the nth day before the current day. Here we calculate *score* based on downloads for the past four days, weighted by recency of downloads [4], [39]. The maximum number of apps in the chart is defined by $N_{MaxTopChart}$.

4) User Agents Browse and Download Apps: Within each user, *daysElapsed* is incremented. If *daysElapsed* exceeds *daysBtwBrowse*, *daysElapsed* is reset to zero and the user browses the app store via the new apps chart, top apps chart, and keyword search. Only apps not previously downloaded are examined. To determine if a new app should be downloaded, the feature grid **F** of the app is compared to the preference grid **P** of the user. If all the nonzero features of an app match the preferences of a user, then the user will download that app. For example, in Fig. 3, the user will download App 1 but not App 2. Upon downloading an app, the user has a probability P_{Rating} of rating the app in the app store. If the app is infectious, the user will inform its network of friends who will examine the features of the app in the next timestep and will download the app if the features match their preferences.

5) Increase Agent Population: The number of user and developer agents in the ecosystem are increased for the next timestep, using (2) and (3), respectively.

AppEco is implemented in C++. The code may be requested via email. This version of AppEco extends previous work [1]–[4] with the addition of: users being able to provide ratings and being influenced by ratings, developers having capability, apps having quality, users having preference intensity, a more balanced mutation for all developer strategies, updated population growth equation to fit six years rather than three years, and calibration from new data provided by survey of 10 000 users [5], see the next section.

IV. MODEL CALIBRATION

We collected the following iOS data over a period of six years, from the start of the iOS ecosystem in July 2008 (Q4 2008) until the end of June 2014 (Q3 2014).

1) *Number of iOS Developers:* This is derived from the number of worldwide iOS developers month on month as compiled by Gigaom.[14]

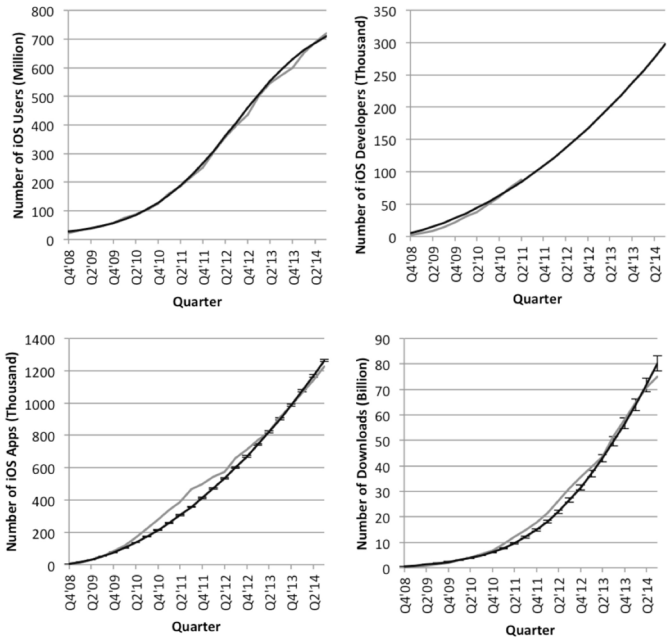

Fig. 5. Actual versus simulated number of iOS users, developers, apps, and downloads, averaged over a hundred runs (gray line is actual, black line is simulated with error bars showing one standard deviation from the mean where the standard deviation is greater than 0).

2) *Number of iOS Apps and Downloads:* These are based on statistics provided in Apple press releases and Apple Events [3].
3) *Number of iOS Users:* The number of iOS users is based on the number of iOS devices (iPod Touch, iPhone, and iPad) sold over time, using sales figures from Apple's quarterly financial data.[15] We then compare the data with Apple's announcements of the number of iOS user accounts to calculate how many users use duplicate devices and also to account for devices that are no longer in use, in order to arrive with the actual number of iOS users.

We calibrate individual values using results we collected from our mobile app user survey [5]. For example, in our survey we found that 48.11% of app users are influenced by ratings and 47.37% of app users provide ratings. Using this and other publicly available data we calibrated AppEco to simulate the iOS app ecosystem. Table III summarizes the calibrated values for the system constants.

The real iOS store contains hundreds of millions of users and tens of billions of downloads. This is computationally infeasible to simulate in terms of memory and time. Thus in AppEco, there is a one-to-many correspondence between user agents and real users, with one user agent representing 10 000 real users. As such, the value of *numFriends* for one user agent is the average *numFriends* for 10 000 real users. There is a one-to-one correspondence between modeled apps and real apps, and between developer agents and real developers.

The actual and simulated number of users, developers, apps, and downloads is shown in Fig. 5. After calibration

[14]http://gigaom.com/apple/infographic-apple-app-stores-march-to-500000-apps/

[15]http://www.apple.com/pr/library/

the growth rates of AppEco closely resembles the growth of the iOS ecosystem, including rates that emerge via interactions, such as the number of apps and downloads [1], [3]. A run of the simulation takes approximately 690 seconds CPU time on a MacBook Pro with a 3 GHz Intel Core i7 processor and 16 GB of 1600 MHz DDR3 memory. After six years (2160 timesteps assuming 30 days a month), the model typically contains more than 290 000 developer agents, 1 250 000 apps, 70 000 user agents (corresponding to 700 million real users), and 7.7 million downloads (corresponding to 77 billion real downloads).

V. Baseline AppEco

A. Objective and Setup

In order to investigate how the dynamics of developer populations affect fitness, we must first establish the baseline behavior of AppEco. In the baseline, the first four developer strategies (S0 innovator, S1 milker, S2 optimizer, and S3 copycat) are used by an equal proportion of developers, and strategy S* flexible is never used. An equal proportion of strategies S0, S1, S2, and S3 is used in the baseline because it enables the comparison of the relative performance of each strategy and enables comparison when the proportions differ in later experiments. AppEco was run for 2160 timesteps (corresponding to six years in the real world, assuming 30 days a month) with the settings described in Section IV. For each timestep, developers in the ecosystem were randomly assigned strategies S0, S1, S2, or S3 in equal proportions.

The following metrics were used to evaluate each strategy.

1) *Average Downloads per App (AvgDl):* For each strategy, *AvgDl* is the total number of downloads received by all developers using that strategy divided by the total number of apps that they have developed.
2) *Top 20 Total Downloads (Top20TotDl):* For each strategy, *Top20TotDl* is the proportion of developers using the strategy that is in the top 20 developers who have received the highest total number of downloads.
3) *Zero Downloads (ZeroDl):* The proportion of developers per strategy who have received zero downloads for all their apps. The lower the value, the better the strategy.
4) *Fitness:* For each app, users are "surveyed" to find out if they would download the app (does its features match their preferences). Apps are then grouped by the experience level of their developers when they were developed: if the app is the developer's first app, then the experience level is 1, if it is the developer's second app, then the experience level is 2, and so on. For each strategy, the *fitness* of the strategy at experience level L is defined as

$$fitness_L = \frac{avgDl_L}{numUsers} \tag{5}$$

where $avgDl_L$ is number of potential downloads as reported by the surveyed users for all the apps produced by developers at experience level L divided by the number of apps in that experience level, and *numUsers* is the number of surveyed users.[16] $Fitness_L$ ranges from 0 to 1. The higher the value, the higher the fitness of the strategy (and the better the strategy can make apps that receive more downloads).

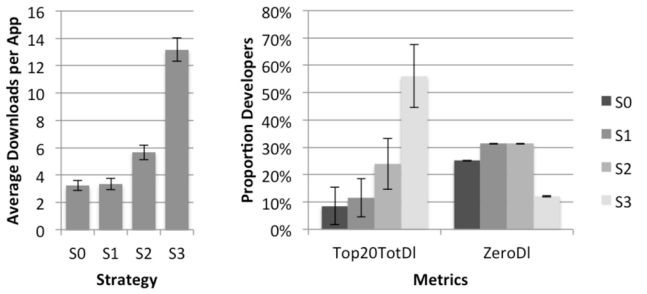

Fig. 6. (a) Average downloads per app (*AvgDl*). (b) Proportion of developers in top 20 total downloads (*Top*20*TotDl*) and zero downloads (*ZeroDl*). Error bars show one standard deviation from the mean.

ANOVA is used to compare between strategies where homogeneity of variances could be assumed [40]. Levene's test is used to indicate whether homogeneity could be assumed. If the outcome of Levene's test is significant ($p < 0.05$), then homogeneity could not be assumed and Welch ANOVA is used. *Post hoc* analyses were also conducted for the statistically significant ANOVA tests. Specifically, when homogeneity could be assumed, Tukey's HSD tests were conducted on all possible pairwise contrasts. Where homogeneity could not be assumed, Games-Howell *post hoc* tests were used [40]. All statistical analyses were conducted using IBM's SPSS statistical analysis package.[17]

The baseline experiment was repeated 100 times, with the results averaged over the 100 runs. To enable comparison with this baseline, all subsequent experiments in later sections also use the same settings, number of runs, and statistical analyses unless specified otherwise.

B. Results

S3 copycat is the most successful strategy in baseline AppEco, receiving the highest *AvgDl* and *Top20TotDl*, and the lowest *ZeroDl*[18] (Fig. 6), with a significant difference compared to the other strategies ($p = .000$) (supplementary material Table 1). This is followed by S2 optimizer, which receives the second highest *AvgDl* and *Top20TotDl*, with a significant difference compared to the other strategies ($p = .000$) (supplementary material Table 1). Although the S0 innovator performs the worst with the lowest *AvgDl* and *Top20TotDl*, it performs second best in terms of *ZeroDl*. S1 and S2 perform the worst in terms of

[16]In this paper, we measure fitness in terms of the number of downloads. This is the standard measure of success used by app stores, which directly determines the payment received by developers. It could be argued that an app with a worldwide market of 20 users is very successful if it is used by all 20 users, however, data for market shares of different applications is not available and such a measure would overcomplicate the model.

[17]http://www-01.ibm.com/software/uk/analytics/spss/

[18]The success of copycats is well known in the real world–several Angry Birds copycats have risen high in the top apps chart and many successful apps in the real world copy the features of apps that are already successful.[8]

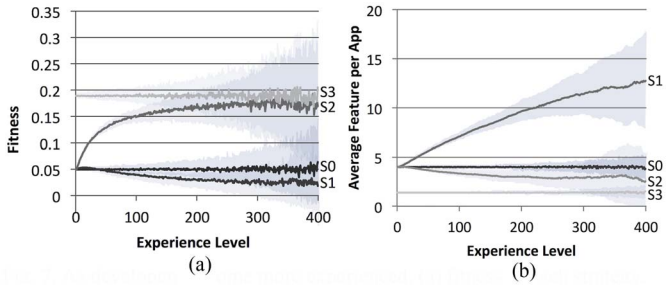

Fig. 7. (a) Fitness of each strategy, and (b) average number of features per app, as developers become more experienced. One standard deviation from the mean is shown as translucent gray areas behind each mean. Later data is sparser as fewer developers create large number of apps, resulting in more noise.

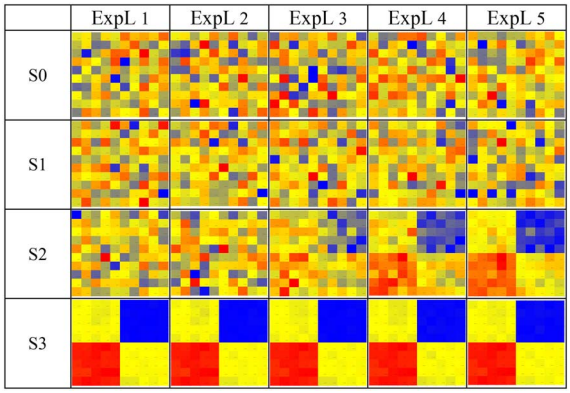

Fig. 8. Heat map showing average app features over experience level (*ExpL*) per developer strategy. Colors change from red to yellow to blue depending on the number of features. Red indicates many features and blue indicates very few features.

ZeroDl, with no significant differences between them (Fig. 6) (supplementary material Table 1). S1 and S2 are both strategies where developers produce new apps based on their existing apps. When developers have no information (or ignore the information) about the quality or success of an app, producing a new app based on an existing bad app is more likely to produce another bad app rather than a good app. (This is the case for S1 milker, since milkers modify their most recent app regardless of performance, and it is the case for S2 optimizer when the developers never receive any downloads.)

S2 optimizer is the only strategy that improves in fitness as the developers become more experienced. Fig. 7(a) illustrates that S2 optimizer is the only strategy that shows a classic evolutionary curve. S2 optimizer gradually produces fewer features per app as developers become more experienced [Fig. 7(b)]. This is because developers using the S2 optimizer strategy learn that fewer features result in more downloads because users only download an app when all the features of the app matches their preference, and as a result apps with fewer features will have a higher chance to match more users' preferences and receive more downloads. To analyze further, we record the number of features per app for each strategy as its developers become more experienced, i.e., have higher experience level. We also group the apps by the experience level of their developers when the apps are built, and then aggregate the features of the apps with the same strategy and experience level and average them over 100 runs to produce a feature heat map. We find that S2 optimizer correctly avoids the top right corner of features not desired by users as they become more experienced, while still consistently covering the features desired by the users (Fig. 8). This demonstrates that developers who improve their own apps based on download feedback can increasingly meet the needs of the users.

S3 copycat has a constant high fitness. S3 copycat produces an average of 1.4 features per app [Fig. 7(b)]. Since copycat developers copy from top apps chart, the apps that they copy are likely to have very few features.[19] Also, as S3 copies from the top apps chart, their heat map matches user preferences right from the start (Fig. 8). However, there is no improvement since there is no learning.

[19]This is consistent with the advice from successful app developers–apps with fewer features have a higher chance of being downloaded, see http://www.zdnet.com/blog/mobile-news/the-common-error-of-overloading-mobile-apps-with-features/5686

S0 innovator has a constant low fitness. Developers using the S0 innovator strategy consistently produce apps with four features regardless of experience level [Fig. 7(b)]. This is because with P_{Feat} being 0.04 (Table III), a random app with 100 cells has on average four features. The heat map shows that S0 innovator produces features randomly regardless of developer experience (Fig. 8).

Interestingly, S1 milker decreases in fitness as developers become more experienced. The heat map illustrates that similar to S0 innovator, S1 milker also produces features randomly regardless of developer experience (Fig. 8). However, the number of features per app for S1 milker increases as developers become more experienced [Fig. 7(b)]. This is due to the three types of mutation in AppEco: 1) add; 2) move; and 3) remove a feature. Adding and moving a feature is always possible, but only features that already exist can be removed; the net result is that apps have a greater probability of gaining features. This models the real-world effect of "feature bloat". This ever-increasing number of features has an adverse effect on the fitness because users only download an app when all the features of the app matches their preference, and apps with more features will have a lesser chance to match a user's preference and as a result it will receive fewer downloads.

The baseline AppEco has a limited form of death as some developers become dormant over time in the model. However, it could be argued that apps also may die as they become obsolete or are removed by their developers or the app store. To evaluate the effect of app death, we ran AppEco with the same settings except that every timestep, 1 out of 10 000 apps is removed. We also tested a second form of app death weighted in favour of more successful apps: every timestep, if the app has downloads, there is 1 out of 100 000 chance it is killed, if it has no downloads, there is 1 out of 10 000 chance it is killed.

The results for both forms of app death showed that the performance of each strategy is the same as seen in the baseline system, with no significant differences. The results for later experiments are also unaffected by app death, so for simplicity it is not included in the AppEco model. The AppEco app

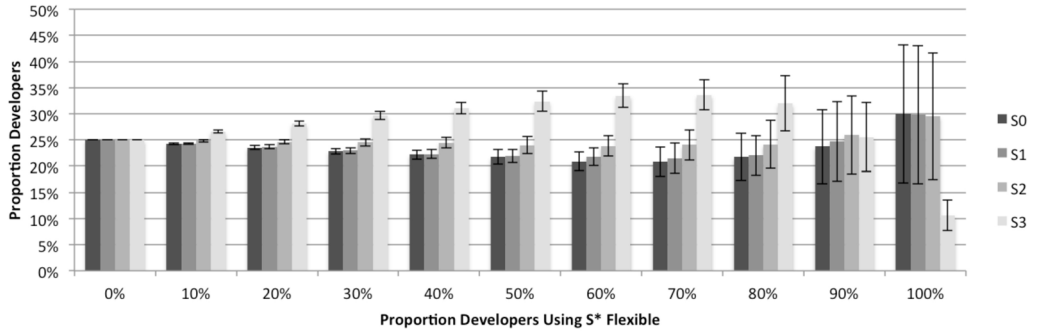

Fig. 9. Proportion of developers in each strategy as the proportion of developers that change strategies increases from S* = 0% to S* = 100%. Error bars show one standard deviation from the mean.

growth rates that were calibrated to match real iOS growth rates therefore implicitly include the losses caused by app removals.

VI. Experiment 1: Is There an Equilibrium for Strategies and How Does it Influence Fitness?

A. Objective and Setup

The baseline established the relative performance of each developer strategy when the population consisted of equal proportions of each strategy. However, this is unrealistic. In reality, app developers are free to choose which strategy they use (modeled as the S* flexible strategy in AppEco), just as different species of an ecosystem are free to become more or less numerous with respect to each other. Thus, the first experiment investigates the question, when developers choose their strategy, *is there an equilibrium for developer strategies in AppEco and how does it influence fitness?*

Some strategies such as S3 copycat act like parasites/predators on the others, stealing their downloads. In a natural ecosystem, this might lead to a stable cyclic predator–prey behavior. Alternatively, in cultural evolution with producer–scrounger dynamics, there might be a stable equilibrium between the two types [33]. In AppEco, where populations continuously increase (and where only a proportion of the app population remains visible) it is unclear what kinds of stability, if any, are possible. If a strategy were more effective then it might quickly dominate all others; less effective strategies might become a tiny minority. However, in many ecosystems, successful strategies for minorities cannot be maintained for majorities [41]. It is therefore of great interest to study how the proportion of developers using each strategy varies over time, and how this affects fitness.

To investigate whether equilibria exist for developer strategies in AppEco, we varied the proportion of developers using the S* flexible strategy from 0% (where there are a fixed number of developers using the first four strategies as described by the baseline AppEco) to 100% (where all developers can choose which of the four strategies to use).

We measure fitness using *avgFitness* as

$$avgFitness = \frac{\sum_{L=1}^{400} fitness_L}{400} \tag{6}$$

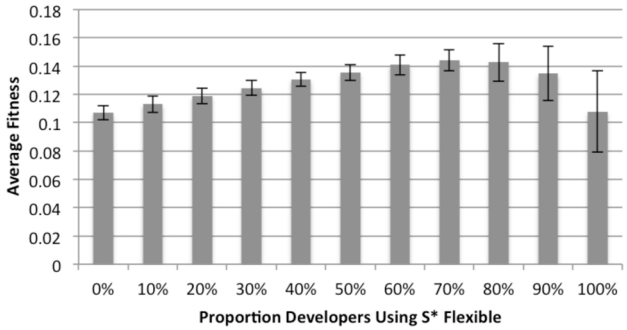

Fig. 10. Average fitness (6) as the proportion of developers using S* flexible increases from 0% to 100%. Error bars show one standard deviation from the mean.

where L is experience level of developers and $fitness_L$ is calculated using (5). The maximum experience level is capped at 400 because developers rarely develop more than 400 apps. We also measure the proportion of developers for each strategy at timestep $t = 2160$. We use these measures for all the subsequent experiments in the paper.

B. Results

The results show that there is no clear equilibrium for strategies S0, S1, and S2. Fig. 9 shows the variability of the results. As more developers choose strategies, the variance of strategy proportions becomes higher across runs. However, as more developers choose their strategies, S3 copycat is used by a slightly higher proportion of developers compared to S0, S1, and S2—up to about 35% of the population using S3 copycat. The average fitness of all apps in the app store also increases (Fig. 10). The trend seems predictable—more choice should surely result in more choosing the successful S3 copycat strategy, which should improve fitness. However, this trend then reverses dramatically when more than 70% of developers choose their strategies. When 100% choose, S3 copycat is consistently and by far the least commonly chosen strategy. It seems surprising and counterintuitive that the overwhelmingly successful S3 copycat falls to about 10% of the population (with comparatively lower standard deviation) when all developers choose their strategies.

There was a significant effect of all proportions of developers using S* flexible on the resulting proportion of developers

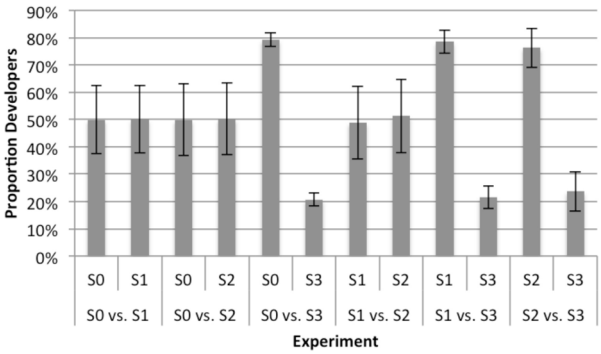

Fig. 11. Proportion of developers in each strategy when two strategies compete. Error bars show one standard deviation from the mean.

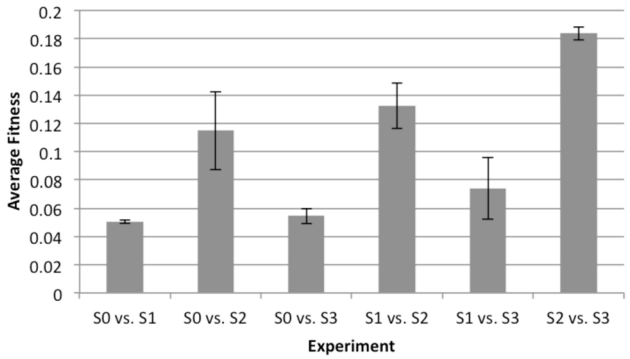

Fig. 12. Average fitness (6) when different pairs of strategies compete.

for each strategy ($p < .05$), except for when S* is 90% ($p = .130$) (supplementary material Table 2).

Post-hoc tests showed that the differences in proportions of strategies between copycat S3 and the other three strategies S0, S1, and S2 (measured pairwise) were significant ($p < .05$) for all of the results, except when S* = 90%. However, the proportions of strategies S0, S1, and S2 were not always significantly different from each other: S0 and S1 were usually not significantly different ($p > .05$), except for S* = 20% and S* = 60%, and when S* = 90% or above, S2 ceased to be significantly different from S0 and S1 ($p > .05$) (supplementary material Table 2).

Enabling more developers to choose their strategies also has a very clear effect on fitness (Fig. 10). Fitness increases gradually from S* = 0% to S* = 70% and then decreases dramatically from S* = 80% to S* = 100%. The standard deviation for the fitness is similar from S* = 0% to S* = 70% and then increases dramatically from 70% onward. There was a significant effect of the proportion of developers using S* flexible on fitness, $F(10, 433.99) = 354.60$, $p = .000$ (supplementary material Table 6). *Post-hoc* tests showed significant differences between all pairs of consecutive groups ($p < .05$), except for S* = 60% compared to S* = 70%, and S* = 70% compared to S* = 80%. It is interesting to note that the fitness for S* = 0% and S* = 100% is also not significantly different (supplementary material Table 6).

In summary, the first experiment has shown that AppEco exhibits instability in terms of the relative numbers of each strategy used by developers. Surprisingly, when averaged across all runs, when more developers choose their strategies there seemed to be no significant consistent winning or losing strategies for S0, S1, and S2, despite the fact that S0 and S1 are random and S2 exploits individual learning. In contrast S3 copycat is almost always significantly different, whether in larger or smaller proportions compared to the other strategies.

Despite the predatory/parasitic nature of S3 copycat, there is no obvious cyclic behavior. Fitness of apps is highest when 70% of developers choose, with S3 copycat having the highest proportion. However, when S* flexible is used by 100% of developers, S3 copycat makes up only 10% of the population. This unexpected finding appears to illustrate a producer–scrounger equilibrium between the first three strategies (producers) and copycat (scrounger) [33].

VII. Experiment 2: What Are the Attractors in the Strategy State Space of the App Store and Do They Help Improve Fitness?

A. Objective and Setup

While average fitness is highest when 70% of developers choose their strategies, in real life, all developers are free to choose their strategy. Thus to understand the stability of developer strategies when S* = 100% in more detail, the second experiment asks: *What are the attractors in the strategy state space of the app store, and do they help improve fitness?* When 100% of developers have the freedom to change their strategy, are some strategies always likely to be chosen more often than others by developers, and if so, why?

1) Two Strategies Compete: We compare the results when S* has the choice of only two strategies, pitting each of the four against each other: [S0 versus S1], [S0 versus S2], [S0 versus S3], [S1 versus S2], [S1 versus S3], and [S2 versus S3]. As comparisons are made between two strategies, *t*-tests are used [40].

2) Delay Strategy Change: We look at isolation effects, allowing time for the different strategies to evolve independently before S* is used. To achieve this we compare different versions of AppEco where S* is used 100% of the time and vary the delay before S* strategy is used, from $d = 0$ (right at the start) to $d = 360$ (after a year).

B. Results

1) Two Strategies Compete: When two strategies compete, the only attractor in the space is low S3 copycat. For [S0 versus S3], [S1 versus S3], and [S2 versus S3], the proportion of S3 copycat developers is consistently low (approximately 20%) (Fig. 11), and the difference with the other competing strategy is significant in each case ($p = .000$). When strategies S0, S1, and S2 compete with each other, the proportions of developers in each strategy are equal (approximately 50:50) with no significant differences ($p > .05$) (supplementary material Table 3).

Average fitness is higher when S2 is involved (Fig. 12). There was a significant effect of competing strategies on fitness, $F(5, 239.77) = 16283.59$, $p = .000$. *Post-hoc* tests showed significant differences between all pairs of groups ($p < .05$): [S0 versus S1] compared to [S0 versus S2], [S0 versus S1] compared to [S1 versus S2], and so on

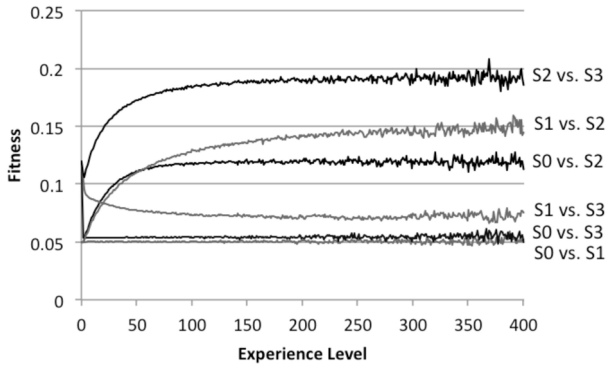

Fig. 13. Fitness (5) for two strategies competing. S1 versus S3 (milker versus copycat) is the only pair that has decreasing fitness as experience level increases.

TABLE IV
AVERAGE STANDARD DEVIATIONS FOR DEVELOPER PROPORTIONS FOR ALL STRATEGIES S0 TO S3 FOR EACH MODEL CHANGE IN EXPERIMENT 3 (SEE SUPPLEMENTARY MATERIAL FIG. 1 FOR STANDARD DEVIATIONS FOR DEVELOPER PROPORTIONS FOR INDIVIDUAL STRATEGIES)

	Model Change	Average SD (%)
S Algorithm*	Original	10
	Instant change	16
	No mutation	10
	Instant change & no mutation	16
Browsing Mechanism	Top apps chart & keyword search	13
	Top apps chart & new apps chart	26
	Top apps chart only	28
Chart Size	Top apps chart only	10
	Top apps chart & new apps chart	6

(supplementary material Table 6). To analyze further, Fig. 13 shows the fitness for two strategies competing over experience level. The results indicate that the evolutionary S2 optimizer is clearly the superior strategy in terms of converging to apps with higher fitness—the three pairs containing S2 all produce apps with higher fitness. The optimal combination of strategies to maximize fitness is [S2 versus S3], producing an average fitness of 0.184 (SD = 0.005). (Separate experiments with strategy S2 alone produce a lower average fitness of 0.157 (SD = 0.005). This difference in fitness is significant $t(198) = -41.11$, $p = .000$.)

2) Delay Strategy Change: The pairwise strategy competitions showed there is a built-in equilibrium for low S3 copycat in AppEco. When we isolate the strategies by adding a delay before developers can choose their strategies, the results show that we are able to remove this equilibrium, changing it from low S3 with significant differences compared to S0, S1, and S2 ($p = .000$), with a delay of zero, to high S3, still with significant differences compared to S0, S1, and S2 ($p = .000$), with a delay of 360, see Fig. 14 (supplementary material Table 4). The standard deviation for the proportion of S3 is very small when $d = 0$ and large when $d = 360$.

Fig. 15 shows the strategy state space, plotting all runs where $d = 0$, $d = 180$, and $d = 360$. The attractor toward S3 = 0.1 (10% proportion) is clearly visible when $d = 0$ (Fig. 15 left column). When $d = 180$ and $d = 360$, there is a weak attractor toward lower values of S0, S1, S2, and higher values of S3 (Fig. 15 middle and right columns).

Thus the results show that S3 copycat requires other strategies to produce good apps before it can achieve success. When developers can change strategy straight away ($d = 0$), S3 is disadvantaged because it cannot copy apps until it has something to copy. As a result, from the start, there will be a low proportion of S3 developers, leading to fewer new developers choosing the S3 strategy for the remainder of the run. However, isolating the separate strategies by delaying the time before developers choose strategies, enables S3 to establish itself as a successful strategy.

There was a significant effect of delaying strategy change on fitness, $F(12, 500.36) = 28.13$, $p = .000$, but *post-hoc* tests showed no significant differences between all pairs of groups next to each other ($p > .05$) (supplementary material Table 6). Finally, Fig. 16 illustrates that the fitness increases as the value of d increases from 0 to 150, and then stabilizes from $d = 150$ onward. This shows that the app store produces apps that please users more, when S3 copycat is used by more developers.

In summary, the second experiment established that the only stable states in strategy space are for S3 copycat. When the time before changing strategies is low, developers using the S3 copycat strategy have nothing to copy and so they quickly choose other strategies. When the time before changing strategies is high and strategies are initially isolated, copycats are able to exploit many existing good apps and will become more numerous in the population (but always less than 50%). This has the result of enabling the production of apps with higher fitness than were seen in Experiment 1. In all cases the strategies S0, S1, and S2 seem to have no consistent competitive advantage over each other. There are no equilibria or obvious attractors in strategy state space for these strategies, with the result that any combination of strategy proportions is possible at any time during the run. This is despite the fact that the combination of S2 and S3 on their own provides the highest fitness overall.

VIII. EXPERIMENT 3: WHAT IS THE RELATIONSHIP BETWEEN STABILITY AND FITNESS?

A. Objective and Setup

With these results in mind, the final experiment investigates the question: *What is the relationship between stability and fitness?* Would a simplified or less realistic model that reduces randomness and increases feedback to developers improve stability and help convergence to apps with higher fitness? Answering this question will enable us to pinpoint the source of the unstable strategy dynamics and determine if the instability helps developers to produce fit apps. We investigate the question by making progressive model changes in order to induce stability in AppEco.

1) S Algorithm:* In the default AppEco, developers using strategy S* start with their own strategy (S0 to S3) and only change strategies after they have uploaded their first app. Also, each developer has a 0.99 probability of randomly selecting an app from the top apps chart and change strategy to be the same

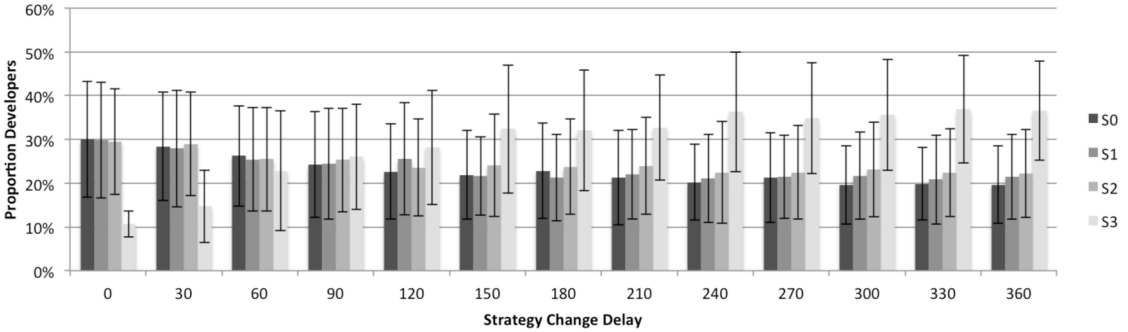

Fig. 14. Average proportion of developers at timestep 2160 as start time of strategy change is delayed from 0 (no delay) until 360. Error bars show one standard deviation from the mean.

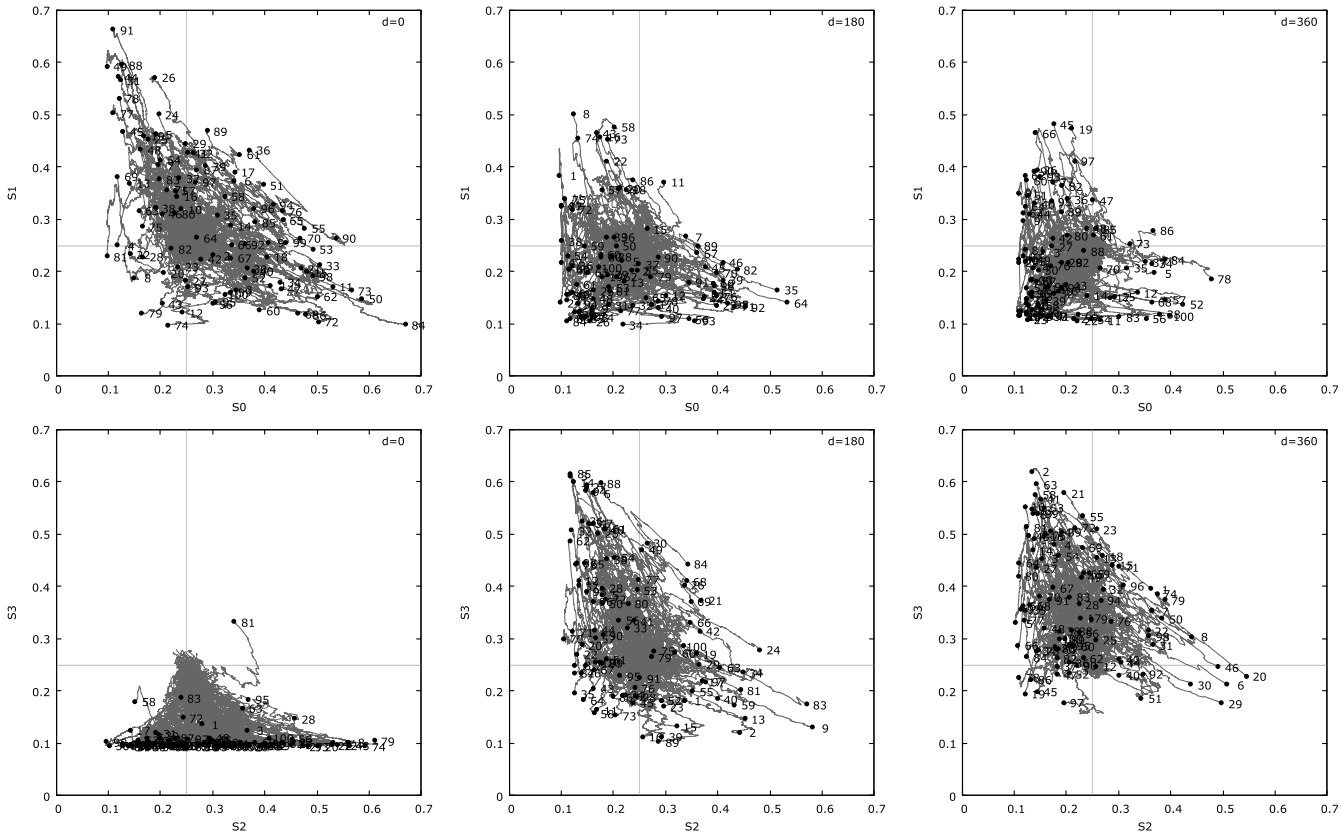

Fig. 15. Strategy state space showing the proportion of each strategy in the population for each run. For clarity the 4D space is plotted as pairs S0 against S1 and S2 against S3, where each line represents one run of the system. The start of each line is (0.25, 0.25) meaning each strategy has a 25% proportion of the population and each point numbered n is the ending point for run n. Results are shown for experiments where delays $d = 0$, $d = 180$, and $d = 360$.

as the developer of the selected app, and a 0.01 probability of selecting a random strategy. Does reducing randomness in strategy use and selection improve stability and what is the resultant effect on fitness? Here we investigate the following variations of S*.

1) *Instant Change:* Developers copy other developers' strategies straight away.
2) *No Mutation:* We remove strategy mutation such that developers always randomly select an app from the top apps chart and change strategy to be the same as the developer of the selected app.
3) *Instant Change and No Mutation:* Combination of both.

2) Browsing Mechanism: In the default AppEco, users browse for apps using all three mechanisms: 1) top apps chart; 2) new apps chart; and 3) keyword search. Does restricting the browsing methods used by users to find apps reduce randomness and improve stability and what is the resultant effect on fitness? Here we use the S* algorithm with instant change and no mutation, and investigate the results when users use the following.

1) Top apps chart and new apps chart.
2) Top apps chart only (note when $t = 0$, the top apps chart is empty because apps have not had any downloads yet, so users look at new apps chart).

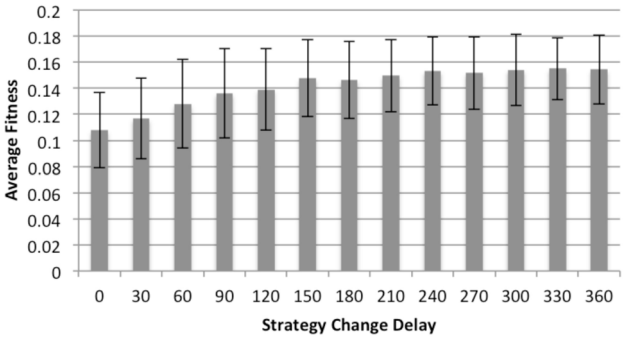

Fig. 16. Average fitness (6) as start time of strategy change is delayed. Error bars show one standard deviation from the mean.

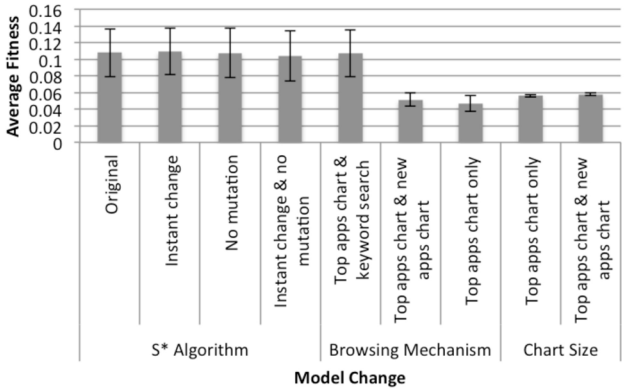

Fig. 17. Average fitness (6) for each model change in Experiment 3. Error bars show one standard deviation from the mean.

3) Chart Size: In the default AppEco, the top apps chart contains the top 50 apps in the store ($N_{MaxTopChart} = 50$), and the new apps chart contains a random sampling ($P_{OnNewChart} = 0.001$) of 40 new apps ($N_{MaxNewChart} = 40$) (Table III). This experiment investigates if providing more complete information about apps in the store improves stability and improves fitness. Here we use the S* algorithm with instant change and no mutation, without keyword search, and set $N_{MaxNewChart}$ to 1 400 000 and $N_{MaxTopChart}$ to 1 400 000. 1 400 000 is more than the number of apps in the store at $t = 2160$, so it ensures that the charts are not constrained. We investigate the results when users use the following.

1) Top apps chart only (note when $t = 0$, the top apps chart is empty because apps have not had any downloads yet, so users look at new apps chart).
2) Top apps chart and new apps chart.

B. Results

For all types of model changes, although the proportion of S0, S1, or S2 may occasionally dominate populations in individual runs, the average proportion of strategies across all runs is approximately 30% for S0, S1, and S2 (with no significant differences between them, $p > .05$), with the proportion of S3 always being close to zero (supplementary material Table 5). The changes in model never change this relationship between S0, S1, S2, and S3, however, they can stabilize the proportions of each strategy during individual runs, as seen by a reduction in average standard deviation, see Table IV. S3 is the only strategy to have a significant difference on the proportion of developers ($p < .05$) (supplementary material Table 5). More detailed results follow.

1) S Algorithm:* Figs. 17 and 18 show the results for modifying the S* algorithm in AppEco. We summarize the findings below.

1) *Instant Change:* When developers immediately copy other developers' strategies, standard deviation of the results increases from 10% to 16%, and the proportion of S3 copycat decreases. This is consistent with the findings of Experiment 2, where reducing the delay before changing strategy reduces the resulting proportion of S3.
2) *No Mutation:* The average standard deviation is the same as the original S* algorithm. The standard deviation for S3 copycat reduces to near zero.
3) *Instant Change and No Mutation:* The average standard deviation increases from 10% to 16%. Combining both provides high standard deviation for S0, S1, S2, and very low proportions of S3 with low standard deviation.

In summary, modifying the S* strategy to reduce randomness has two opposite effects: instant change decreases stability, while no mutation in S* partially increases stability. As shown by Fig. 17, fitness is similar to AppEco with default settings, so it seems that stochasticity within strategy S* neither helps nor hinders fitness.

2) Browsing Mechanism: Figs. 17 and 18 show the results for modifying the browsing mechanism in AppEco. We summarize the results below.

1) *Top Apps Chart and Keyword Search:* When the new apps chart is not available to users, it is evident that the strategies S0, S1, and S2 appear to maintain relatively equal proportions, with no strategy completely dominating a run.
2) *Top Apps Chart and New Apps Chart:* When keyword search is disabled, individual runs show that individual strategies (S0, S1, or S2) now dominate the population more often than before.
3) *Top Apps Chart Only:* When users only choose apps based on their ranking in the top apps chart, considerable randomness has now been eliminated. Strategies in individual runs often settle at stable proportions (in many cases with S0, S1, or S2 completely dominating the population).

In summary, reducing randomness in the user browsing mechanisms has a clear stabilizing effect of strategy proportions during individual runs. This is because reducing the randomness in user browsing mechanisms also reduces the chances of changing the proportion of developers and their strategies on the top apps chart, and as a result, reducing the butterfly effect that causes the overall proportion of developers to change.

The relative proportion of different strategies is usually unique for each run. This is consistent with the finding of Experiment 2 where it was shown that there are no overall winning strategies and one losing strategy of S3. Interestingly, fitness is dramatically reduced when keyword search is absent.

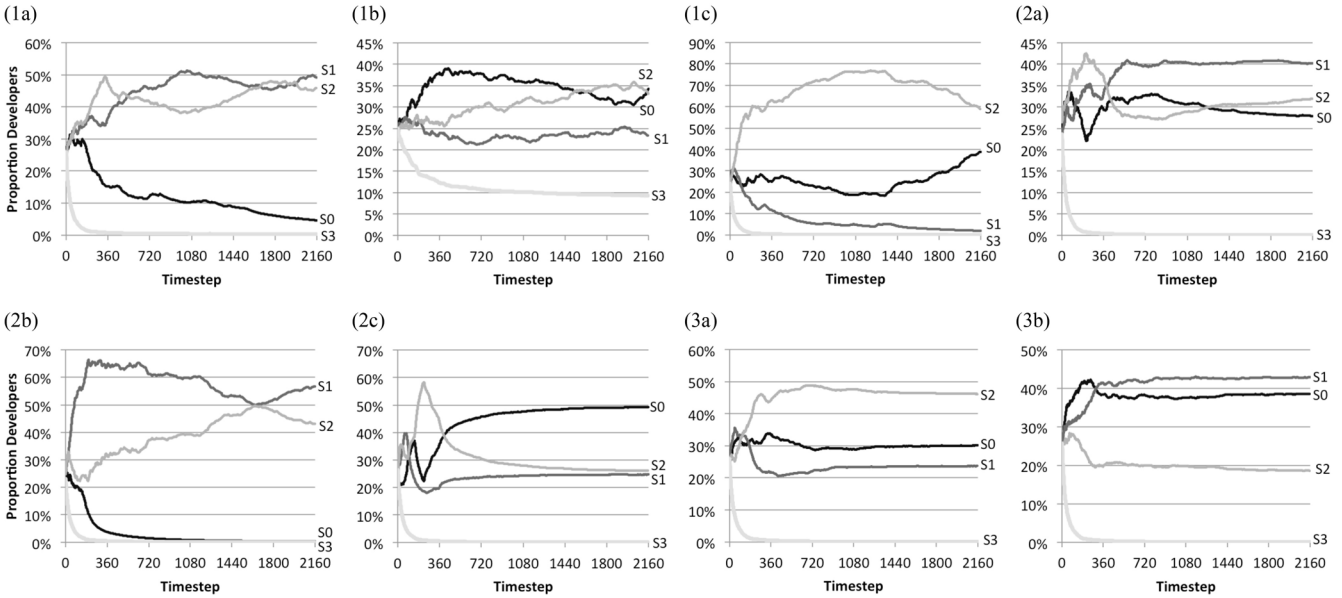

Fig. 18. Representative runs for S* algorithm. (1a) Instant change. (1b) No mutation. (1c) Instant change and no mutation. Representative runs for Browsing Mechanism. (2a) Top apps chart and keyword search. (2b) Top apps chart and new apps chart. (2c) Top apps chart only. Representative runs for Chart Size. (3a) Top apps chart only. (3b) Top apps chart and new apps chart. When averaged across all runs, strategies S0, S1, and S2 are used in equal proportions with no significant differences between them, and S3 is always in the minority.

It seems that the randomness (and corresponding instability) of keyword search is essential in AppEco. Users need to be able to find any app in the store; if they only see and download those few apps visible in the top charts there is excessive selection pressure and the app store prematurely converges on apps with poor features.

3) Chart Size: Figs. 17 and 18 show the results for modifying the chart size in AppEco. We summarize the results below.

1) *Top Apps Chart Only:* When users are able to see the relative success of more apps in the store, there is a dramatic decrease in variance across runs. Individual runs also show that strategies S0, S1, and S2 settle at consistent proportions, and no single strategy tends to dominate the population.
2) *Top Apps Chart and New Apps Chart:* When users can see the relative success of more apps in the store plus also all new apps introduced, variance decreases further. This is now the most stable version of the model, with S0, S1, and S2 coexisting at stable proportions in each run.

In summary, instability in AppEco can be reduced by (and thus stability caused by) removing random mutation from the S* strategy, removing keyword search, and providing information about more apps to users. Fitness is still low because of the lack of keyword search, but slightly improved because of the additional information provided by having larger charts. Variance of fitness is reduced.

Overall for Experiment 3, there was a significant effect of model change on fitness, $F(8, 352.25) = 216.10$, $p = .000$. *Post-hoc* tests showed significant differences ($p < .05$) between pairs of groups involving the following: browsing mechanism—top apps chart and new apps chart, browsing mechanism—top apps chart only, chart size—top apps chart only, and chart size—top apps chart and new apps chart. It seems that by removing keyword search, the standard deviation reduces and the differences become significant (supplementary material Table 6).

S3 copycat "scrounger" will always be a minority strategy compared to the three producer strategies. S0, S1, and S2 can coexist at different, stable proportions, although it is not possible to predict which of the three producer strategies will be more or less numerous in any single run. However, increasing stability has no beneficial effect (and frequently a significant detrimental effect) on fitness, because of high selective pressures caused by the lack of keyword search and because copycat is often forced out of the population entirely. It seems that instability in AppEco in terms of developer strategy proportions and selection by users is beneficial to fitness. Stability improves the consistency of results, but dramatically worsens the quality.

IX. Conclusion

This paper asked whether population dynamics within an ecosystem can drive search in a useful way, and if so, what kinds of dynamics might exist that could be useful. To address these questions, we focused on a man-made evolutionary ecosystem: the commercial software environment known as the app store. AppEco is a model of a human ecosystem that has two unique features: 1) it exhibits dynamics reminiscent of those seen in biological ecosystems and 2) it also can be said to perform optimization: the ecosystem as a whole tries to find a range of apps with features that best match the many diverse user preferences in order to maximize downloads. Using AppEco, we investigated the stability and instability within app store dynamics and how it affects fitness.

Baseline runs with AppEco established that with fixed proportions of strategies, S3 copycat produces the fittest apps that receive the most downloads and which contain the fewest features. S2 is the only strategy able to optimize its apps toward

user preferences and consistently improve fitness. Three experiments were then run where developers were free to choose their own strategies, in order to investigate causes of stability in AppEco and their effect on fitness.

Experiment 1 asked: *is there an equilibrium for developer strategies in the app store and how does it influence fitness?* The results showed that in one respect there is no equilibrium. Different strategies may dominate in any single run, and this is often highly unstable with other strategies taking over at any time during a run. However, in another respect there is a clear equilibrium: as more developers choose strategies, copycats become more numerous (comprising up to a third of the population), but when 100% developers choose strategies, S3 copycat consistently becomes least numerous (comprising just 10% of the population). This finding mirrors the findings of other models in theoretical biology. As stated by Barnard [43] and supported by Barta and Giraldeau [42], "scrounger strategies do best when they are rare and worst when common relative to producer strategies." It is clear that in the app store, the S0 innovators, S1 milkers, and S2 optimizers are all producer strategies—they do not copy information from others. S3 copycat is a scrounger—it copies information about successful apps from others [33]. In fact, copycats provide an important way for the app store to optimize fitness. The results highlighted the importance of copycats—overall fitness improves as more developers use the copycat strategy. The explorative S0, S1, and S2 are enhanced by the exploitative S3. The results also showed that as more developers choose their strategies, the significant difference between the evolutionary optimizer S2 and the random searches of S0 and S1 becomes lost.

Experiment 2 then asked: *what are the attractors in the strategy state space of the app store and do they help improve fitness?* The results showed clearly that although S0 and S1 are random and S2 is an evolutionary optimizer, all major dynamics resulted from the producer–scrounger effect. The model has a clear attractor in strategy state space for low proportion of S3 copycat; no other attractors were evident. The only way to disrupt this attractor was to isolate the strategies by delaying the introduction of S*. Again, this confirms the findings in theoretical biology [33]: AppEco naturally falls into a state where the proportion of copycats is low and stable compared to the other strategies. S0, S1, and S2 combined were naturally always more dominant in the population compared to S3, but individually none were consistently more dominant than each other. The experiment also confirmed that fitness increased as copycats were used more, and the best fitness was observed when only S2 optimizer competed with S3 copycat. Given that the producer–scrounger dynamics are akin to exploration/exploitation in search, these findings may reveal opportunities for evolutionary computation in the future, with new operators playing the role of copycat in multispecies evolution or ensemble learning.

The extreme variability of relative proportions of S0, S1, and S2 in the population was then studied. In the final experiment we asked: *What is the relationship between stability and fitness?* The results showed that reducing randomness in the S* strategy, removing keyword search, and providing information about more apps to users, resulted in stable and converged proportions of all strategies, with S0, S1, and S2 sharing roughly equal proportions and S3 having little or no proportion. Yet even in this state of reduced noise, none of the three producer strategies S0, S1, or S2 converged to consistent or predictable proportions in the population, and when keyword search was removed and S3 copycat was reduced to near zero, overall fitness suffered. It seems that for AppEco, increased stability of S0, S1, and S2 is not beneficial. Improved consistency of fitness is of little benefit if the resulting fitness is dramatically worse, thus it is clear that AppEco relies on stochasticity to help improve fitness. The only useful stability as far as maximizing fitness is concerned is the stable proportion of S3 copycat with all of the other strategies.

It is difficult to measure the proportion of copycat developers in real life in order to corroborate these findings in the iOS App Store, as the judgment of app similarity is subjective. App stores such as iOS App Store and Google Play reject apps that are obvious clones, but there is clear evidence that copycat developers persist,[20] with some tantalizing data points available. For example, in March 2014 following the wildly successful number one app Flappybird, there were on average 60 clones of the app submitted to the app store each day for a few days, with many becoming best-selling apps in their own right.[21] While this may sound high, at its peak it was less than 5% of the total apps submitted each day during the same period,[22] suggesting that while copycats were very active and successful, they were in the minority in the app store on this date, as predicted by AppEco.

In the real world there are likely to be many more types of randomness than exist in AppEco. However, this work has shown that even in this unusual ecosystem, which has massive growth of populations and many different types of interacting strategy, there are clear and explicable dynamics that can help improve overall fitness. The app store is not filled with predators and prey. It is filled with many producers who have no predictable advantage over each other, and a few important scroungers. Fitness is maximized only when they coexist together in stable proportions of producer and scrounger. A sustainable ecosystem may self-regulate this proportion, keeping scroungers a steady minority in the population.

If we were to design the best app ecosystem for optimizing the fitness of apps (rather than modeling a real app store) this work suggests that a combination of just two self-regulating developer strategies provide the very best results: evolutionary optimizer and copycat. It is possible that producer–scrounger algorithms based on such ecosystem dynamics will make useful new evolutionary algorithms in the future. The second author is investigating optimizers based on ecosystems [44], [45].

[20] http://www.macworld.co.uk/feature/iosapps/how-app-store-got-taken-over-by-copycats-3512145/

[21] http://www.pocketgamer.co.uk/r/iPhone/Flappy+Bird/news.asp?c=57880

[22] According to: http://www.statista.com/statistics/258160/number-of-new-apps-submitted-to-the-itunes-store-per-month/

REFERENCES

[1] S. L. Lim and P. J. Bentley, "App epidemics: Modelling the effects of publicity in a mobile app ecosystem," in *Proc. 13th Int. Conf. Synth. Simulat. Living Syst. (ALIFE)*, East Lansing, MI, USA, 2012, pp. 202–209.

[2] S. L. Lim and P. J. Bentley, "From natural to artificial ecosystems," in *Proc. Frontiers Nat. Comput.*, York, U.K., 2012, p. 15.

[3] S. L. Lim and P. J. Bentley, "How to become a successful app developer? Lessons from the simulation of an app ecosystem," in *Proc. Genet. Evol. Comput. (GECCO)*, Philadelphia, PA, USA, 2012, pp. 129–136.

[4] S. L. Lim and P. J. Bentley, "Investigating app store ranking algorithms using a simulation of mobile app ecosystems," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Cancun, Mexico, 2013, pp. 2672–2679.

[5] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden, "Investigating country differences in mobile app user behavior and challenges for software engineering," *IEEE Trans. Softw. Eng.*, vol. 41, no. 1, pp. 40–64, Jan. 2015.

[6] P. A. Whigham and G. B. Fogel, "Ecological applications of evolutionary computation," in *Ecological Informatics*. Berlin, Germany: Springer, 2006, pp. 85–107.

[7] D. Morrall, "Ecological applications of genetic algorithms," in *Ecological Informatics*. Berlin, Germany: Springer, 2003, pp. 35–48.

[8] A. J. Watson and J. E. Lovelock, "Biological homeostasis of the global environment: The parable of daisyworld," *Tellus B*, vol. 35, no. 4, pp. 284–289, 1983.

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems*. 2nd ed. Cambridge, MA, USA: MIT Press, 1992.

[10] T. S. Ray, "An approach to the synthesis of life," in *Artificial Life II*. Redwood City, CA, USA: Addison-Wesley, 1991, pp. 371–408.

[11] N. Minar, R. Burkhart, C. Langton, and M. Askenazi, *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations*, Santa Fe Inst., Santa Fe, NM, USA, 1996.

[12] J. M. Epstein and R. Axtell, *Growing Artificial Societies: Social Science From the Bottom Up*. Washington, DC, USA: Brookings Inst. Press, 1996.

[13] R. G. Palmer, W. B. Arthur, J. H. Holland, B. LeBaron, and P. Tayler, "Artificial economic life: A simple model of a stockmarket," *Phys. D Nonlin. Phenom.*, vol. 75, nos. 1–3, pp. 264–274, 1994.

[14] D. McFadzean, D. Stewart, and L. Tesfatsion, "A computational laboratory for evolutionary trade networks," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 546–560, Oct. 2001.

[15] B. LeBaron, "Empirical regularities from interacting long- and short-memory investors in an agent-based stock market," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 442–455, Oct. 2001.

[16] S. Martinez-Jaramillo and E. P. K. Tsang, "An heterogeneous, endogenous and coevolutionary GP-based financial market," *IEEE Trans. Evol. Comput.*, vol. 13, no. 1, pp. 33–55, Feb. 2009.

[17] K. Kinoshita, K. Suzuki, and T. Shimokawa, "Evolutionary foundation of bounded rationality in a financial market," *IEEE Trans. Evol. Comput.*, vol. 17, no. 4, pp. 528–544, Aug. 2013.

[18] K. Takadama, T. Terano, and K. Shimohara, "Nongovernance rather than governance in a multiagent economic society," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 535–545, Oct. 2001.

[19] J.-C. Chen, T.-L. Lin, and M.-H. Kuo, "Artificial worlds modeling of human resource management systems," *IEEE Trans. Evol. Comput.*, vol. 6, no. 6, pp. 542–556, Dec. 2002.

[20] H.-Y. Quek, K. C. Tan, and H. A. Abbass, "Evolutionary game theoretic approach for modeling civil violence," *IEEE Trans. Evol. Comput.*, vol. 13, no. 4, pp. 780–800, Aug. 2009.

[21] D. B. Knoester, H. J. Goldsby, and P. K. McKinley, "Genetic variation and the evolution of consensus in digital organisms," *IEEE Trans. Evol. Comput.*, vol. 17, no. 3, pp. 403–417, Jun. 2013.

[22] K. Izumi and K. Ueda, "Phase transition in a foreign exchange market-analysis based on an artificial market approach," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 456–470, Oct. 2001.

[23] T. Tassier and F. Menczer, "Emerging small-world referral networks in evolutionary labor markets," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 482–492, Oct. 2001.

[24] G. Fagiolo, G. Dosi, and R. Gabriele, "Matching, bargaining, and wage setting in an evolutionary model of labor market and output dynamics," *Adv. Complex Syst.*, vol. 7, no. 2, pp. 157–186, 2004.

[25] D. W. Bunn and F. S. Oliveira, "Agent-based simulation—An application to the new electricity trading arrangements of England and Wales," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 493–503, Oct. 2001.

[26] J. Nicolaisen, V. Petrov, and L. Tesfatsion, "Market power and efficiency in a computational electricity market with discriminatory double-auction pricing," *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 504–523, Oct. 2001.

[27] H. Ishibuchi and N. Namikawa, "Evolution of iterated prisoner's dilemma game strategies in structured demes under random pairing in game playing," *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp. 552–561, Dec. 2005.

[28] S. Y. Chong and X. Yao, "Multiple choices and reputation in multiagent interactions," *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 689–711, Dec. 2007.

[29] P. Tino, S. Y. Chong, and X. Yao, "Complex coevolutionary dynamics—Structural stability and finite population effects," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 155–164, Apr. 2013.

[30] K. M. Sim and Y. Wang, "Evolutionary asymmetric games for modeling systems of partially cooperative agents," *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp. 603–614, Dec. 2005.

[31] S. Samothrakis, S. Lucas, T. P. Runarsson, and D. Robles, "Coevolving game-playing agents: Measuring performance and intransitivities," *IEEE Trans. Evol. Comput.*, vol. 17, no. 2, pp. 213–226, Apr. 2013.

[32] H. Dawid, M. Reimann, and M. Bullnheimer, "To innovate or not to innovate?" *IEEE Trans. Evol. Comput.*, vol. 5, no. 5, pp. 471–481, Oct. 2001.

[33] A. Mesoudi, "An experimental simulation of the 'copy-successful-individuals' cultural learning strategy: Adaptive landscapes, producer–scrounger dynamics, and informational access costs," *Evol. Human Behav.*, vol. 29, no. 5, pp. 350–363, 2008.

[34] L. Tesfatsion, "Agent-based computational economics: Growing economies from the bottom up," *Artif. life*, vol. 8, no. 1, pp. 55–82, 2002.

[35] L. Tesfatsion. (Mar. 11, 2015). *Agent-Based Computational Economics (ACE)*. [Online]. Available: http://www2.econ.iastate.edu/tesfatsi/aapplic.htm

[36] S. A. Elias, *Origins of Human Innovation and Creativity*, vol. 16. Boston, MA, USA: Elsevier, 2012.

[37] R. I. M. Dunbar, "Neocortex size as a constraint on group size in primates," *J. Human Evol.*, vol. 22, pp. 469–493, Jun. 1992.

[38] S. E. Kingsland, *Modeling Nature: Episodes in the History of Population Ecology*. Chicago, IL, USA: Univ. Chicago Press, 1995.

[39] M.-C. Lanfranchi, B. Benezet, and R. Perrier, *How to Successfully Market your iPhone Application*, faberNovel, Paris, France, 2010.

[40] A. P. Field, *Discovering Statistics Using SPSS*. Los Angeles, CA, USA: SAGE, 2009.

[41] R. Axelrod and W. D. Hamilton, "The evolution of cooperation," *Science*, vol. 211, no. 4489, pp. 1390–1396, 1981.

[42] Z. Barta and L.-A. Giraldeau, "The effect of dominance hierarchy on the use of alternative foraging tactics: A phenotype-limited producing-scrounging game," *Behav. Ecol. Sociobiol.*, vol. 42, no. 3, pp. 217–223, 1998.

[43] C. Barnard, *Parasitism and Host Behaviour*. New York, NY, USA: CRC Press, 2002.

[44] M. T. Adham and P. J. Bentley, "An ecosystem algorithm for the dynamic redistribution of bicycles in London," presented at the Proc. 10th Int. Conf. Inf. Process. Cells Tissues (IPCAT), San Diego, CA, USA, 2015.

[45] M. T. Adham and P. J. Bentley, "An artificial ecosystem algorithm applied to static and dynamic travelling salesman problems," presented at the Proc. IEEE Symp. Ser. Comput. Intell. (SSCI) Int Conf. Evol. Syst. (ICES), Orlando, FL, USA, 2014.

Soo Ling Lim received the Bachelor of Software Engineering degree (First Class Hons.) from Australian National University, Canberra, ACT, Australia, in 2005 and the Ph.D. degree in computer science and engineering from University of New South Wales, Sydney, NSW, Australia, in 2011.

She is an Honorary Research Associate with the Department of Computer Science, University College London, London, U.K. She was an SAP Consultant with Computer Sciences Corporation, Canberra; a Software Engineer with CIC Secure, Canberra; an Assistant Professor with National Institute of Informatics, Tokyo, Japan; and a Research Associate with University College London and Bournemouth University, Poole, U.K. Her research interests include software requirements elicitation and prioritization, software development, mobile app development, and agent-based modeling.

Dr. Lim was a recipient of the Dean's Prize from the Australian National University, and the Australian Computer Society Prize in 2005.

Peter J. Bentley received the Bachelor of Science degree (Hons.) in computer science (artificial intelligence) from the University of Essex, Colchester, U.K., in 1993, and the Ph.D. degree in evolutionary computation applied to design from the University of Huddersfield, U.K., in 1996.

He is an Honorary Reader and a Senior College Teacher with the Department of Computer Science, University College London (UCL), London, U.K.; a Collaborating Professor with Korean Advanced Institute for Science and Technology, Daejeon, Korea; a Freelance Writer; and an App Developer. He runs the Digital Biology Group, UCL. He has published over 250 scientific papers, edited books entitled *Evolutionary Design by Computers, Creative Evolutionary Systems*, and *On Growth, Form and Computers*, and authored a book entitled *The Ph.D Application Handbook* and popular science books entitled *Digital Biology*, *The Book of Numbers*, *The Undercover Scientist*, and *Digitized*. His research interests include modeling and bio-inspired computation, such as agent-based modeling, evolutionary algorithms, computational development, and other complex systems, applied to diverse applications, such as software engineering, design, mobile wireless devices, security, and novel computation.

Fuyuki Ishikawa received the Bachelor of Science, Masters, and Ph.D. degrees in information science and technology from University of Tokyo, Tokyo, Japan, in 2002, 2004, and 2007, respectively.

He is an Associate Professor with the National Institute of Informatics, Tokyo. He is also a Visiting Associate Professor with University of Electro-Communications, Chofu, Japan, and an Adjunct Lecturer with University of Tokyo, Waseda University, Tokyo; the Tokyo Institute of Technology, Tokyo; and the Japan Advanced Institute of Science and Technology, Nomi, Japan. His research interests include services computing, such as service description, selection and composition, and software engineering, such as requirements analysis, formal methods, and self-adaptation. He has published over 70 papers in the above areas, some of which are in high-ranking conferences such as International World Wide Web Conference, International Conference on Web Services, and International Conference on Service-oriented Computing.