İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GEZGİN SATICI PROBLEMİNİN HADOOP ÜZERİNDE ÇALIŞAN PARALEL GENETİK ALGORİTMA İLE ÇÖZÜMÜ

YÜKSEK LİSANS TEZİ

Harun Raşit ER

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

HAZİRAN 2013

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GEZGİN SATICI PROBLEMİNİN HADOOP ÜZERİNDE ÇALIŞAN PARALEL GENETİK ALGORİTMA İLE ÇÖZÜMÜ

YÜKSEK LİSANS TEZİ

Harun Raşit ER
(504091515)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Nadia ERDOĞAN

HAZİRAN 2013

İTÜ, Fen Bilimleri Enstitüsü'nün 504091515 numaralı Yüksek Lisans Öğrencisi **Harun Raşit ER**, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"GEZGİN SATICI PROBLEMİNİN HADOOP ÜZERİNDE ÇALIŞAN PARALEL GENETİK ALGORİTMA İLE ÇÖZÜMÜ"** başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Prof. Dr. Nadia ERDOĞAN** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Yrd. Doç. Dr. Sanem SARIEL** İstanbul Teknik Üniversitesi	.....
	Yrd. Doç. Dr. Yunus Emre SELÇUK Yıldız Teknik Üniversitesi	.....

Teslim Tarihi : **02 Mayıs 2013**
Savunma Tarihi : **06 Haziran 2013**

ÖNSÖZ

Günümüzde büyük veri setlerini işlemek için yüksek maliyetli süper bilgisayarların kullanımı yerine standart bilgisayarlardan oluşan bir ağ üzerinde dağıtık veri işleme yönteminin kullanılması önem kazanmıştır. Bunun amacı ise, kaynak kullanımını optimize etmek ve maliyeti düşürmektir. Bu çalışmada, dağıtık programlamanın en güncel örneklerinden biri üzerinde çalışmak mutluluk ve heyecan verici bir deneyim olmuştur.

Tez çalışmam boyunca her konuda desteğini benden esirgemeyen, olumsuz durumlarda beni yeniden çalışmaya yönlendiren, saygıdeğer tez danışmanım Prof. Dr. Nadia ERDOĞAN'a teşekkürlerimi sunarım.

Bugün bu noktada bulunmamda en büyük pay sahibi olan anneme, tez çalışmamı tamamlamamı benden daha fazla isteyen babama ve tez konusundaki her türlü yakınmalarıma katlanan kardeşlerim Ali ve Mümin'e sonsuz teşekkürler.

Mayıs 2013

Harun Raşit Er
(Bilgisayar Mühendisi)

İÇİNDEKİLER

Sayfa

KISALTMALAR

GA **:** Genetic Algorithm / Genetik Algoritma
GSP **:** Gezgin Satıcı Problemi
HDFS **:** Hadoop Distributed File System
JVM **:** Java Virtual Machine
NP **:** Non-Polinomial
PGA **:** Parallel Genetic Algorithm / Paralel Genetik Algoritma
SGA **:** Sequential Genetic Algorithm
TSP **:** Traveling Salesman Problem

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

GEZGİN SATICI PROBLEMİNİN HADOOP ÜZERİNDE ÇALIŞAN PARALEL GENETİK ALGORİTMA İLE ÇÖZÜMÜ

ÖZET

MapReduce, günümüzde birçok büyük ölçekli firmanın büyük miktardaki verileri işlemek için kullandıkları bir dağıtık programlama altyapısıdır. *MapReduce* altyapısının asıl amacı, büyük miktardaki verileri paralel olarak işlemektir. Ancak son zamanlarda işlemci-yoğun programlar için de kullanılmaya başlanmıştır.

Google tarafından ortaya atılan bu programlama modeli - isminden de anlaşılacağı gibi - "map" ve "reduce" fonksiyonlarından oluşmaktadır. Kullanıcı bir *MapReduce* programı yazmak istediğinde en azından bu iki fonksiyonu gerçeklemek zorundadır. Map fonksiyonu dışarıdan anahtar/değer ikilisi alır, bunları işler ve çıktı olarak yeni bir anahtar/değer ikilisi verir. Reduce fonksiyonu ise; bir anahtar değeri ve o anahtar değerine sahip tüm değerlerin listesini girdi olarak alır. Listedeki veriler üzerinde işlem yaptıktan sonra çıktı olarak bir anahtar/değer ikilisi verir.

MapReduce programlama modeli, büyük miktardaki verileri işlemek için yüksek maliyetli süper bilgisayarlar ve veri tabanı sunucuları yerine sıradan bilgisayarlardan oluşan bir ağ kullanır. Ağ üzerindeki bilgisayarlar üzerinde map ve reduce fonksiyonları paralel olarak çalıştırılır. *MapReduce* mimarisi ağ üzerinde veriyi taşımak yerine verinin bulunduğu makinelere ilgili map veya reduce fonksiyonunu gönderir. Böylece ağ üzerinde oluşacak bant genişliği problemini ortadan kaldırır.

Hadoop, MapReduce altyapısının ücretsiz ve açık-kaynak gerçeleştirimidir. *Hadoop* kullanıcıya iki yapı sunmaktadır. Bunlardan birisi kullanıcının dağıtık program geliştirmesini ve çalıştırmasını sağlayan *MapReduce* yapısıdır. Diğeri ise; geliştirilen *MapReduce* programının girdi ve çıktılarının saklandığı dağıtık dosya sistemi olan *Hadoop* Dağıtık Dosya Sistemi'dir (*Hadoop* Distributed File System - HDFS).

HDFS üç parçadan oluşur. *Namenode*, HDFS üzerindeki verilerin yönetimini yapmaktan sorumludur. *Secondary Namenode*, adından anlaşılacağı üzere, *Namenode*'un yedeği olarak çalışmaktadır. *Datanode* ise, gerçek verilerin saklandığı bölümdür. *Hadoop* sisteminde *Namenode* ve *Secondary Namenode* birer adet bulunur ve master makine üzerinde çalışır. *DataNode* ise bir veya daha fazla sayıda olabilir ve slave makineler üzerinde çalışır.

Hadoop MapReduce yapısı ise; *Jobtracker* ve *TaskTracker* olmak üzere iki ana parçadan oluşur. *JobTracker Hadoop* işlerinin çalıştırılmasını koordine eder. Kullanıcıdan çalıştırılacak olan işi alır ve onun düzgün bir şekilde çalıştırılıp sonlandırılmasını sağlar. *Tasktracker* kullanıcınının yazmış olduğu map ve reduce fonksiyonlarını çalıştırır. *Hadoop* sisteminde bir adet *JobTracker* bulunur ve master makine üzerinde çalışır. *TaskTracker* bir veya birden fazla olabilir ve slave makineler üzerinde çalışır.

Genetik Algoritma, optimizasyon problemlerinin çözümünde sıklıkla kullanılan sezgisel arama algoritmasıdır. Genetik algoritma doğal hayattaki en iyi olanın hayatta

kalması prensibine göre çalışır. Buna göre her jenerasyonda en iyi bireyler hayatta kalır ve bir sonraki jenerasyon için yeni bireylerin oluşturulmasına katkı sağlarlar.

Genetik algoritma doğal evrim sürecinden esinlerek geliştirilmiştir. Buna göre ilk olarak bir başlangıç popülasyonu oluşturulur. Bu ilgili problem için oluşturulan rastgele çözümler kümesidir. Daha sonra popülasyondaki her bireyin ne kadar iyi çözüm oldukları hesaplanır. Sonraki adımda bireyler ne kadar iyi olduklarına göre eşleşme için seçilir. Seçilen bireyler, aralarında gen değişimi gerçekleştirerek yeni bireyleri oluştururlar. Son olarak yeni bireyler belirlenen bir olasılıkla mutasyona uğrarlar. Bu şekilde yeni bir jenerasyon ortaya çıkar. Yeni jenerasyondaki bireylerin herhangi birisinin sonlanma koşulunu sağlayıp sağlamadığı kontrol edilir. Eğer sonlanma koşulu sağlanıyorsa algoritma sonlanır. Aksi taktirde, algoritma önceki adımları tekrar eder.

Genetik algoritmalar her ne kadar kabul edilebilir bir zaman içerisinde optimum çözüme yakın çözümler üretseler de bazı problemler içermektedir. Bunlardan en önemli ikisi; uygunluk değerinin hesaplanması ve algoritmanın yerel optimum değere takılmasıdır. Özellikle popülasyondaki bireylerin sayısının fazla olduğu durumlarda bireylerin uygunluk değerinin hesaplanması çok fazla zaman alabilmektedir. Bu sorunların çözümü için paralel genetik algoritmalar kullanılır.

Paralel genetik algoritmalar uygunluk değerinin hesaplanma yöntemine göre, tek veya birden çok popülasyon kullanımına göre ve birden fazla popülasyon kullanımı durumunda, bireylerin popülasyonlar arasındaki değişim yöntemine göre sınıflandırılmaktadır. Buna göre genetik algoritmalar üç ana sınıfa ayrılabilir. Bunlar; master-slave modeli, göç alan statik alt-popülasyonlar modeli ve statik kesişen alt-popülasyonlar modelidir.

Göç-alan statik alt-popülasyonlar modelinde birden fazla alt-popülasyon bulunur. Alt-popülasyonlar birbirlerinden bağımsız olarak bireyler üzerinde genetik operatörleri uygularlar. Ve belirlenen aralıklarla göç operatörü yardımı ile bireyler diğer popülasyonlara göç ettirilir. Bu şekilde algoritma çözüm uzayının çok daha büyük bir kısmını taramış olur ve yerel optimum değerine takılma sorununu ortadan kaldırır.

Bu çalışmada *Hadoop* ağı üzerinde genetik algoritmanın paralelleştirilmesi gerçekleştirilmiştir. Paralelleştirme yöntemi olarak ise göç-alan statik alt-popülasyonlar yöntemi kullanılmıştır. Buna göre her alt-popülasyon ayrı bir map/reduce görevi olarak farklı bir bilgisayarda çalışır. Böylece tüm alt-popülasyonlar paralel olarak farklı makinelerde genetik operatörleri bireylere uygular. *Hadoop* işinin tamamlanması sürecinde; ilk olarak tüm map görevleri HDFS'den tüm popülasyonlara ait bireyleri okurlar. Bireyler popülasyon kimliklerine göre reducer fonksiyonuna gönderilir. Buna göre aynı popülasyon kimlik numarasına sahip bireyler aynı reducer tarafından işlenir. Popülasyonlar belirlenen sürede evrim geçirdikten sonra sonuçlar tekrar HDFS'e yazılır ve *Hadoop* işi sonlanır. Bir sonraki *Hadoop* işi ise daha önce yazılan popülasyonları okuyarak başlar ve aynı süreç, sonlanma koşulu sağlanana kadar devam eder.

Gerçekleştirilen genetik algoritmanın test edilmesi için gezgin satıcı problemi kullanılmıştır. Bilindiği gibi gezgin satıcı problemi (GSP) birçok arama yönteminin test edilmesinde kullanılmaktadır. GSP, bağlı bir çizgede tüm noktaların bir kez ziyaret edilerek başlangıç noktasına geri dönen en kısa yolun bulunması problemidir. GSP, NP-zor kombinasyonel optimizasyon problemidir ve bilgisayar bilimlerinde önemli bir yere sahiptir.

Gerçekleştirilen ardışıl genetik algoritma ve daha önceki yapılan çalışmalarda kullanılan genetik algoritmaların sonuçları karşılaştırılmıştır. Ayrıca *MapReduce* ağı üzerinde paralelleştirilen genetik algoritmanın sonuçları ile ardışıl genetik algoritmanın sonuçları karşılaştırılmıştır. Sonuçlara bakıldığında, gerçekleştirilen ardışıl genetik algoritma diğer algoritmalardan daha kısa sürede daha iyi sonuçlar üretmiştir. Ayrıca paralel genetik algoritma, problem boyutunun büyük olduğu durumlarda ardışıl algoritmadan daha kısa sürede daha iyi sonuçlar üretmiştir.

PARALLEL GENETIC ALGORITHM TO SOLVE TRAVELING SALESMAN PROBLEM ON HADOOP CLUSTER

SUMMARY

MapReduce is a distributed framework that large-scaled companies use to handle large data sets. The main objective of this framework is to process huge data set in a parallel manner using a cluster of commoduty computers. It is also used in CPU-intensive computations in recent years.

Google has proposed the *MapReduce* model that enables users to develop and run distributed programs easily. *MapReduce* programming model has two main components that are called map and reduce funtions. It is necessity to implement these two functions at least in order to write a *MapReduce* program. Map function reads the input data as key/value pairs. Then it produce an intermediate key/value pairs. Reduce function reads the results of map funtions. It takes a key and a list of values associated with this key as its input data. It performs required operations on the list of values. At the end of the *MapReduce* job, reduce function writes the results to a file as key/value pairs.

MapReduce programming model uses commodity computer cluster instead of large database servers or super computers in order to process data. Map and reduce functions are run in the machines on the cluster in parallel. *MapReduce* model uses data locality. As a result, data is not transferred on network but the map and reduce funtions are sent to the machine where the data is stored instead. This eliminates the network bandwidth problem.

Hadoop is open source implementation of *MapReduce* framework for writing and running distributed applications that process large amounts of data. It provides two main concepts for the users: *MapReduce* programming model and *Hadoop* Distributed Fie System (HDFS). *MapReduce* programming model enables users to develop and run the application. HDFS is a distributed file system on *Hadoop* cluster in order to store input and output data of *MapReduce* program.

HDFS contains three main components. *Namenode* is responsible for managing data on HDFS. *Secondary Namenode*, as the name suggests, runs as a backup for *Namenode*. *Datanode* stores the real data on HDFS. *Hadoop* cluster has only one *Namenode* and one *Secondary Namenode* running on master machine. And one or more *Datanode* running on slave machines on the cluster.

Hadoop MapReduce consists of two main components called *Jobtracker* and *TaskTracker*. *Jobtracker* coordinates the running of *Hadoop* jobs on the cluster. It receives the job file from the user and manage the cluster to run the job consistently and gives the results back to user. *Tasktracker* runs the map and reduce tasks in parallel. *Hadoop* cluster contains only one *Jobtracker* running on master machine and one or more *Tasktracker* running on slave machines.

Genetic algorithm is a heuristic search method that is commonly used in the solution of optimization problems. It uses Darwin's Theory where the only best individuals

survive. According to the theory, the best individuals survive in each generation and contribute to creation of new individuals for the next generation.

Genetic algorithm is inspired from natural evolution. The first step of the algorithm is creation of initial population. The initial population is a set of random solutions for the problem. The next step is to evaluate the fitness of each individual. Then individuals are selected according to their fitness values for crossover. Crossover operator changes some genes between selected individuals and constructs the new individuals. The last step is mutation which changes some genes of the new individuals randomly in order to conserve the diversity of solutions. As a result a new generation is obtained. Before the next iteration, the algorithm checks whether any of the individuals in the population satisfies termination condition. If it does the algorithm terminates, otherwise the algorithm goes into the next iteration where fitness calculation, selection, crossover and mutation operators are applied to new population.

One of the drawbacks of the genetic algorithm is that it can get stuck in local optimum. In order to avoid this situation, we propose preventing consanguineous marriage. Our policy in this respect is selecting two individuals that have different gene orders over a particular percent for the crossover operation. Setting a percentage value too high prevents the algorithm from converging. Otherwise, if the percentage value is minimized too much, it has no effect on avoiding from local optima. After some experimentation, we have chosen the percentage value as %20. Therefore, if two parents have similarity over %80, they cannot be chosen for the crossover.

Eventhough the genetic algorithm produces near optimal solutions in a reasonable time, there are some problems to be addressed. The two of most important problems are fitness evaluation time of individuals and getting stuck in local optima as mentioned above. Fitness evaluation can be very time consuming and sometimes impossible especially when the population size is large. If the genetic algorithm gets stuck in local optima, it will never find the global optimum solution. In order to overcome these problems, parallel genetic algorithm (PGA) is used.

The calculation of fitness evaluation, the use of single or multiple population, in case of multiple populations, the way individual exchange is carried out are some of the criteria according which PGAs are classified. PGAs are classified into Master-slave, multiple populations with migration and multiple populations without migration.

Master slave parallelization method includes one master machine and one or more slave machines. Master machine manages the whole genetic algorithm phases. This method contains one population. While fitness calculation is processed by slave machines in parallel, the other phases of genetic algorithm are carried out by master machine sequentially.

This method can be implemented as synchronous or asynchronous. In synchronous method, master machine waits all slave machines to complete fitness calculations. After all slaves send the results, the master machine continues in order to complete the other phases of genetic algorithm. The slowest slave machine determines the performance of the algorithm in this case. In asynchronous method, master machine does not wait for all slave machines to complete their tasks. Master machine applies the genetic operators for those individuals that are received recently and again waits for some individuals to be sent by slave machines.

Static populations without migration model uses more than one populations. Exchange of traits between sub-populations is done by individuals that reside in overlapping areas. These individuals belong to more than one population and they compete in different sub-populations. The genetic operators should be applied sycnhronously on these individuals that reside in overlapping areas.

Static population with migration model, which is used in this study, contains more than one population. All sub-populations apply genetic operators independent of each other. And some individuals are transferred to other populations via migration operator at specified intervals. As a result, sub-populations search the space in different directions and the algorithm avoids getting stuck into local optima.

Parallelization of genetic algorithm on *MapReduce* framework is implemented on *Hadoop* cluster in this study. Multiple population with migration method is used as parallelization method. Each sub-population evolves in a different map/reduce task on different machines. As a result sub-popuations apply genetic operators to their individuals in parallel. The first step of a *Hadoop* job is that map functions read all individuals from HDFS. Individuals are grouped according to their population identifiers and sent to reducer funtions. Reducer receives a population identifier and a list of individuals sharing the population identifier. Reducers apply genetic operators to individuals and populations evolve until migration phase. At he end of reduce funtion, individuals are written back to HDFS. The key point is to write indviduals that will migrate to other populations with different population identifiers. So they will compete within different populations in the next *Hadoop* job. The next *Hadoop* job starts to read individuals from HDFS and continue running as the previous job. This process continues until the termination condition is met.

Travelling Salesman Problem (TSP) is used to test the algorithm implemented. TSP is a common bechmark for testing optimization algorithms. TSP can be defined as finding shortest possible tour which visits every node exactly once and returns to starting node in a connected graph. TSP is an important NP-Hard combinatorial opmimization problem that is studied in computer science.

Implemented sequential genetic algorithm is compared with other implemetations studied before. *MapReduce* parallel genetic algorithm is also compared with the sequential genetic algorithm. Compared to other implementations, the implemented sequential genetic algorithm always finds better results and takes less time. The comparison of *MapReduce* PGA and sequential GA shows that *MapReduce* PGA always finds better results than the sequential one. While sequential GA takes less time for small size of problems, *MapReduce* PGA exhibits better perfomance when the problem size increases. Input/Output operations and Java Virtual Machine (JVM) creation for *Hadoop* jobs dominates the run time of the parallel genetic algorithm for small size of problems. When the problem size increases, I/O operations and JVM creation takes the same time and *MapReduce* PGA run time does not change significantly. But sequential genetic algorithm run time increases dramatically.

Convergence analysis for the sequential genetic algorithm and the previous works is carried out. Sequential genetic algorithm shows better convergence than the previous implementations. But the problem is that, the randomizing factor of the genetic algorithm developed is not so much. So it can get stuck into local optimum solution. To overcome this problem, *MapReduce* PGA is used. Because of multiple population, the algorithm can seach the solution space at different directions and it always get away local optimum.

The main objective of this study is to introduce *MapReduce* framework for the next studies. And also, it tests the performance of *Hadoop* for cpu-intensive algorithms. Recent studies practice on parallelization of GA on *MapReduce* framework also, but they all concentrate on parallelizing fitness function. In this study, multiple population parallelization method is adopted to *MapReduce* infrastructure. And it is shown that multiple population PGA can best fit to *MapReduce* model and it can be scaled to a larger cluster without any effort.

1. GİRİŞ

Enerji kaynaklarının hızla tükendiği ve zamanın çok daha fazla önem kazandığı çağımızda, ortaya çıkan problemlerin çözümünde kullanılacak olan kaynakların optimizasyonu çok büyük önem teşkil etmektedir. Bilgisayar bilimleri bu problemlere çözüm arayan alanlardan birisidir. Gezgin Satıcı Problemi (GSP) [1], bilgisayar bilimleri alanında çalışılan NP-Zor kombinasyonel optimizasyon problemidir [2]. GSP, düğüm noktalarının ve aralarındaki uzaklıkların verildiği bir çizgede, tüm noktaların yalnız ve sadece bir kez ziyaret edilmesi ve başlangıç noktasına geri dönülmesi koşulu ile mümkün olan en kısa yolun bulunması olarak tanımlanır. GSP için birçok kullanım alanı olmakla beraber, bunlardan bazıları; lojistik, planlama, devre tasarımı ve ağ üzerinde paketlerin yönledirilmesidir. Ayrıca GSP, birçok problemin için alt problem olarak karşımıza çıkar, örneğin DNA sıralaması.

GSP çözümü için kesin ve sezgisel birçok çözüm yöntemi sunulmuştur. Kesin çözümlerin en basit olanı olası tüm permutasyonların denenmesidir. Bu çözümün çalışma zamanı *O(n!)*'dir [3]. Ancak bu çözümde çok küçük problem büyüklüğü için bile çözüm zamanı inanılmaz derecede fazla olacaktır. Örneğin 20 noktadan oluşan bir çizgede çözümün çalışma zamanı *O(20!)* olacaktır. Yani 20 noktadan oluşan bir problem için bile yaklaşık $2.5e^{18}$ farklı yolun denenmesi gerekmektedir. Diğer kesin çözüm yöntemleri ise; *"brach and bound"* ve *"brach and cut"* algoritmalarıdır [4].

Kesin çözümlerin yanında birçok sezgisel ve kestirim algoritmaları da önerilmiştir. Sezgisel yöntemler tur oluşturma ve tur iyileştirme olarak iki bölüme ayrılabilir. Tur oluşturma yöntemleri verilen uzaklık bilgilerini kullanarak en kısa turu oluşturmaya çalışır ve tur tamamlandığında elde edilen çözümün iyileştirilmesi ile ilgilenmez. Bu tür çözümlere örnek olarak; en yakın komşu algoritması, aç-gözlü kestirim yöntemi ve ekleme yöntemi verilebilir. Tur iyileştirme yöntemleri ise; problem için bir çözüm turu oluşturulduktan sonra, bazı sezgisel yöntemler ile çözümün iyileştirilmesini öngörür. Benzetilmiş tavlama, 2-opt, 3-opt, k-opt ve genetik algoritmalar sezgisel yöntemlere örnek olarak verilebilir [4].

Genetik Algoritma (GA), GSP probleminin çözümünde kullanılan sezgisel optimizasyon algoritmasıdır. Genetik Algoritma, anlaşılması kolay olması ve etkin sonuçlar üretmesi nedeni ile optimizasyon problemlerinin çözümümde çoklukla tercih edilmektedir. GA, doğal evrim sürecinden esinlenerek oluşturulmuştur; üreme, seçilme ve mutasyon gibi doğal evrim süreçlerini bilgisayar ortamında modelleyerek her nesilde çözümün daha iyiye doğru evrilmesini sağlar. Buna göre, ilk olarak rastgele bireylerden (kromozom, çözüm) meydana gelen bir popülasyon oluşturulur. Daha sonra popülasyondaki tüm bireyler için uygunluk değeri hesaplanır. Hesaplanan uygunluk değerine göre, bireyler çaprazlama işlemi için seçilir. Seçilen bireylerin çaprazlanması ile yeni popülasyon oluşturulur. Son olarak yeni popülasyona mutasyon işlemi uygulanarak bir sonraki nesil için bireyler oluşturulur [5].

Ardışıl Genetik Algoritmalar, birçok alanda birçok problem için çok başarılı çözümler üretmişlerdir. Ancak aşağıda belirtilen durumlar için ardışıl GA yetersiz kalabilmektedir ve bu durumlar için paralel genetik algoritma (PGA) kullanılmaktadır [6].

- Popülasyon büyüklüğünün çok fazla olması gerektiği problemlerde, tüm bireylerin aynı makinede saklanması mümkün olmadığından PGA kullanılması gerekmektedir.
- GA için en fazla zaman gereksinimi duyan aşama popülasyondaki tüm bireyler için uygunluk değerinin hesaplanmasıdır. Zaman kısıtının olduğu durumlarda ardışıl GA yetersiz kalmakta ve PGA kullanılması daha hızlı çözüm elde edilmesini sağlamaktadır.
- Ardışıl GA bazen yerel optimum değerine saplanıp kalmakta ve global optimum değerine ulaşamamaktadır. PGA ise çözüm uzayını aynı anda farklı yönlerde tarayarak global optimum değerine ulaşma olasılığını arttırmaktadır.

PGA her ne kadar zaman anlamında ardışıl GA'dan daha hızlı sonuçlar vermiş olsa da, ardışıl GA'ya göre asıl iyileşme; PGA tek makinede çalıştırılmış olsa bile, aynı anda daha fazla yönde arama yaparak daha kaliteli çözümlere ulaşmasıdır.

GA'nın paralelleştirilmesi sürecinde bazı parametrelerin belirlenmesi gerekmektedir. Bunlar; uygunluk fonksiyonunun hesaplanma şekli, tek popülasyon ya da çoklu alt-popülasyon kullanımı, çoklu alt-popülasyonda bireylerin alt-popülasyonlar

arasındaki değişim şekli ve seçim işleminin global ya da yerel olarak gerçekleştirilmedir [6]. Bu kriterlere bakılarak PGA'lar aşağıdaki gibi gruplanabilir:

- Master-Slave
- Göç operatörlü durağan alt-popülasyonlar
- Birbiri ile kesişen durağan alt-popülasyonlar
- Dinamik alt-popülasyonlar

Master-Slave PGA'da popülasyondaki bireyler için uygunluk değeri hesaplanması slave makineler yardımı ile paralel şekilde yapılırken, diğer tüm GA operatörleri popülasyona ardışıl olarak uygunlanmaktadır. Diğer paralelleştirme yöntemlerinde ise birden fazla alt-popülasyon farklı makinelerde biribirinden bağımsız olarak gelişirken, aralarında bazı bireylerin paylaşılması ile popülasyonlar arasında iletişim sağlanmış olur.

Çoklu alt-popülasyonların kullanıldığı PGA, dağıtık bir ortam üzerinde çalışır. Dağıtık ortamda haberleşme ve gerekli protokollerin tanımlandığı bir alt yapı gerekmektedir. Google tarafından ortaya atılan ve günümüzde sıkça karşımıza çıkan *MapReduce* modeli [7], bilgisayar kümesi üzerinde dağıtık programlama yapılabilmesine olanak sağlamaktadır.

MapReduce mimarisinin çekirdeği *map* ve *reduce* fonksiyonlarıdır [8]. Kullanıcı bu fonksiyonları gerçekleyerek dağıtık bir uygulama geliştirmektedir. *MapReduce* mimarisi bu fonksiyonları dağıtık olarak çalıştırarak veri setlerinin paralel şekilde işlenmesine olanak sağlar. *MapReduce* mimarisi daha çok veri-yoğun işlemler için kullanılmaktadır [9]. Ancak kodlamasının ve anlaşılmasının kolay olması, otomatik olarak hata tespiti yapması ve gerekli hata ayıklama işlemini gerçekletirebilmesi nedeni ile işlemci yoğun uygulamalar için de kullanılmaktadır.

MapReduce modeli için birçok gerçekleştirim mevcuttur. *Hadoop* [10] büyük veri dosyalarının paralel şekilde işlenmesi için kullanılan *MapReduce* yönteminin açık kaynak kodlu gerçekleştirimidir. Bilgisayar kümeleri üzerinde çalışan dağıtık programlar için alt yapı desteği sunar. Kullanıcı etkileşimi gerekmeksizin verilerin ilgili düğüme taşınması ve olası bir donanım sorununda kurtarma işlemlerini otomatik gerçekleştirir. Kullanılan *MapReduce* yönteminde, yazılan program bilgisayar kümesi üzerinde bulunan bir düğümde çalışabilecek küçük parçalara ayrılır. *Hadoop* ayrıca bilgisayar kümesi için yüksek bant genişliği sağlayan "HDFS"

[10] dağıtık dosya sistemini kullanıcıya sunar. Hem *MapReduce* hem de "HDFS" dosya sisteminin kullanımı ile birlikte düğüm hataları otomatik olarak düzeltilir ve kullanıcıya yüksek güvenilirlikli bir sistem sunulur [8].

1.1 Tezin Amacı

Bu çalışmanın amacı genetik algoritmanın *Hadoop* sistemi üzerinde çalışan bir PGA'ya dönüştürülmesi ve geliştirilen PGA yardımı ile TSP probleminin çözülmesi ve sonuçların diğer yapılan çalışmalar ile karşılaştırılmasıdır. İlk olarak ardışıl GA geliştirilmiştir ve TSP probleminin çözümünde kullanılmıştır. Daha sonra *Hadoop* alt-yapısının kurulumu yapılmış ve *Hadoop* üzerinde çalışan PGA geliştirilmiştir. Geliştirilen PGA, TSP çözümü için *Hadoop* alt-yapısı üzerinde çalıştırılmış ve sonuçlar paylaşılmıştır.

1.2 Literatür Araştırması

David E. Goldberg ve arkadaşları genetik algoritmaları paralelleştirme yöntemi olarak *MapReduce* altyapısını kullanmışlardır [8]. Genetik algoritmaların nasıl map ve reduce görevlerine dönüştürülebileceğini, *MapReduce* gibi veri-yoğun bir altyapıda genetik algoritmaların nasıl sonuç vereceğini ve büyük problemlerin çözümünde nasıl kullanılabileceğini sonuç olarak sunmuşlardır.

Bu çözümde; genetik algoritmanın her iterasyonu için bir iş(Job) kullanılmaktadır. Bu nedenle iterasyon sayısı kadar iş çalıştırılmakta olup, her iterasyon sonunda sonuçlar HDFS dosya sistemine yazılmakta ve yeni iterasyon başlangıcında bir önceki yazılan dosyaların okunması işlemi gerçekleştilmektedir.

Map fonksiyonu (görevi) her bireyin uygunluk değerini hesaplar, uygunluk değeri en büyük olan bireyini (popülasyonun en iyi çözümünü) HDFS dosya sistemi üzerinde global bir dosyaya yazar. Ve uygunluk değeri hesaplanan bireyler, çaprazlama ve mutasyon işlemleri için reduce fonksiyonuna (görevine) gönderilir.

Hangi bireyin hangi Reduce görevine gönderileceği işlemi ise, *MapReduce* mimarisinin sunmuş olduğu partitioner metodu ile çözümlenir. Burada kullanılan yöntem her birey'in anahtar değeri için 0 ile reduce görev sayısı arasında bir sayının rastgele belirlenip, bireyin anahar değerine göre ilgili reducer'a gönderilmesidir.

Reduce fonksiyonu, map fonksiyonundan gelen bireyler için çaprazlama ve mutasyon işlemini gerçekleştirir. Ve sonuçları bir sonraki iş(iterasyon) için HDFS dosya sistemine yazar. Çaprazlama için kullanılacak bireylerin belirlenmesi için turnuva yöntemi kullanılmıştır. Buna göre N adet birey rastgele seçilir ve aralarından en iyi iki tanesi çaprazlama işlemine tabi tutulmak üzere belirlenir. Çaprazlama yöntemi olarak uniform çaprazlama yöntemi kullanılmaktadır. Buna göre iki kromozom arasındaki genler belilenen rastgele noktalar arasında %50 oranında değiştirilerek yeni bireyler elde edilir. Oluşan her iki birey de genlerinin yarısını bir atadan diğer yarısını diğer atadan almaktadır.

Ele alınan problem OneMax [8] problemidir. Problem şu şekilde tanımlanabilir; N boyutundaki bir bit listesinde listenin toplamını en büyük değere taşıyacak bitlerin bulunmasıdır. Problemin en iyi çözümü tüm bitlerin "1" olması durumudur. Problem çözümünde 52 bilgisayar ve 416 çekirdek kullanılmıştır. Her bilgisayar 12 GB ram ve 2TB sabit disk alanı içermektedir. Elde edilen sonuçlara bakıldığında algoritmanın 10^5 büyüklüğünde bir problemi çözebildiği ve 10^4 değişken için 220 iterasyon gerçekleştirildiği ve 9 saat süre harcadığı belirtilmektedir.

Fan Yang'ın 2010 yılında yapmış olduğu çalışmada; MPI alt yapısı kullanılarak, paralel genetik algoritmanın Gezgin Satıcı probleminin çözümünde nasıl sonuçlar verdiği gösterilmiştir [11].

Master olarak adlandırılan bilgisayar ilk popülasyonun oluşturulması, seçim, çaprazlama ve mutasyon işlemlerini gerçekleştirmektedir. Slave makineler ise, bireylerin uygunluk değerini paralel olarak hesaplamaktadır. Hesaplanan uygunluk değerleri master makineye geri gönderilmektedir. Tüm bireyler için uygunluk değeri hesaplaması bittiğinde master makine popülasyondaki tüm bireyler için genetik algoritma işlemlerini ardışıl olarak uygulamaktadır.

Bireylerin kodlanmasında premutasyon kodlama yöntemi kullanılmaktadır. Buna göre her şehir için 1'den N'e kadar bir numara belirlenir. Ve bunların bir liste üzeride oluşturduğu sıra çözümü belirtmektedir. Seçim yöntemi olarak rulet tekerleği yöntemi kullanılmaktadır. Çaprazlama yöntemi olarak iki noktalı çaprazlama yöntemi kullanır. Belirlenen iki nokta arasındaki genler birinci atadan alınırken, belirlenen noktalar dışındaki genler ikinci atadan sırasıyla alınır.

1000 nokta içeren TSP problemi için çözüm üretmektedir. Ancak çözüm için harcanan süre ya da çözüm kalitesi ile ilgili herhangi bir bilgi verilmemekte bunun yerine yeni bir makine eklendiğinde çözümdeki hızlanma ve master-slave arasındaki iletişim bilgileri hakkında detaylı grafik bilgileri sunulmaktadır.

Chun-Wei Tsai ve arkadaşlarının Tayvan'da yaptıkları bu çalışmada; paralelleştirme yöntemi olarak iri taneli alt-popülasyonlar yöntemi kullanmaktadır [12]. Her alt-popülasyon ayrı bir iş parçacığı olarak kodlanmıştır. Tüm alt-popülasyonların aynı makine üzerinde çalışması nedeniyle ağ üzerindeki geçikmelerden söz edilemez.

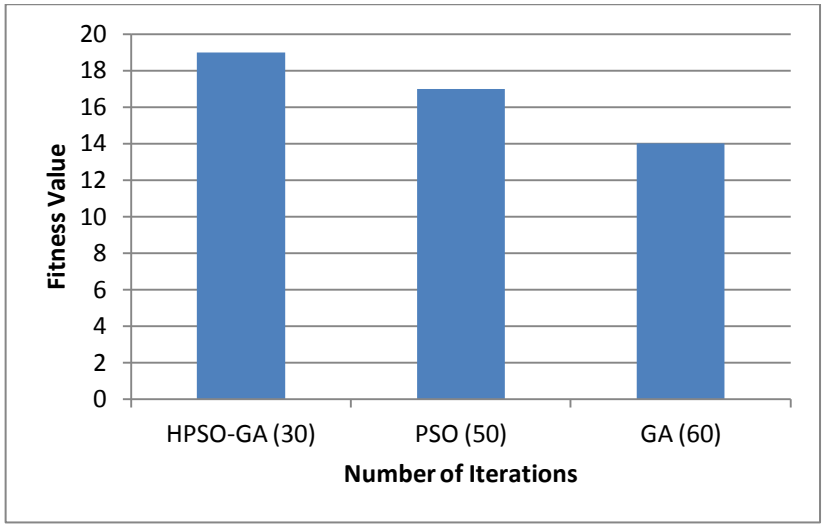
Alt-popülasyonlar kendi içlerinde evrilirler ve belli iterasyon sonunda alt-popülasyonlar arasında bireylerin değişimi gerçekleştirlir. Alt-popülasyon sayısı 4 olarak seçilmiştir. Algoritma şu şekilde tanımlanmaktadır; Örnekleme yöntemi ile başlangıç popülasyonu oluşturulur. Oluşturulan popülasyon N adet alt-popülasyona bölünür. Bundan sonra her alt-popülasyon için standart uygulanan prosedur ise; çaprazlama için en uygun bireyleri seç, bireyler üzerinde çaprazlama ve mutasyon işlemlerini gerçekleştir, tüm bireylerde ortak olan bir pattern varsa bunu belirle ve diğer popülasyonlara bunu gönder. Belirlenen iterasyon sonunda alt-popülasyonlar arasında bireyleri göç ettirilir.

Geliştirilen algoritma tüm popülasyondaki bireyler için belirli bir örüntü oluştuğunda bunu parça çözüm olarak saklamakta ve bu genler üzerinde artık genetik operator uygulamamaktadır. Böylece problemin boyutu sürekli küçülmektedir. Bu da algoritmanın hızlanmasını sağlamaktadır [12]. Son olarak oluşturulan parçalı çözümler birleştirilmekte ve problemin çözümü olarak sunulmaktadır.

G. Sudha Sadasivam ve Dharini Selvaraj yaptıkları çalışmada [13], heterojen görevlerin ağ üzerinde heterojen bir yapıdaki makinelere nasıl dağıtılabileceğini araştırmışlardır. Bunun için *Hadoop* üzerinde paralel olarak çalışan parçacık sürü optimizasyonu (PSO) ve genetik algoritma karışımı hibrit bir yöntem geliştirmişlerdir. PSO ile hızlı bir şekilde çözüm yakınsanırken, GA yardımı ile de yerel optimum çözüme takılma olasılığını ortadan kaldırmayı sağlamışlardır. Map görevleri paralel şekilde parçacıkların hız ve konum bilgilerini günceller, parçacıklar arasında seçim ve çaprazlama işlemlerini gerçekletirir. Reduce taskı ise parçacıkları sıralar, mutasyon işlemini gerçekleştirir ve en iyi bireyi seçer. Çaprazlama operatörü olarak tek noktalı çaprazlama ve mutasyon operatörü olarak segment-mutasyon

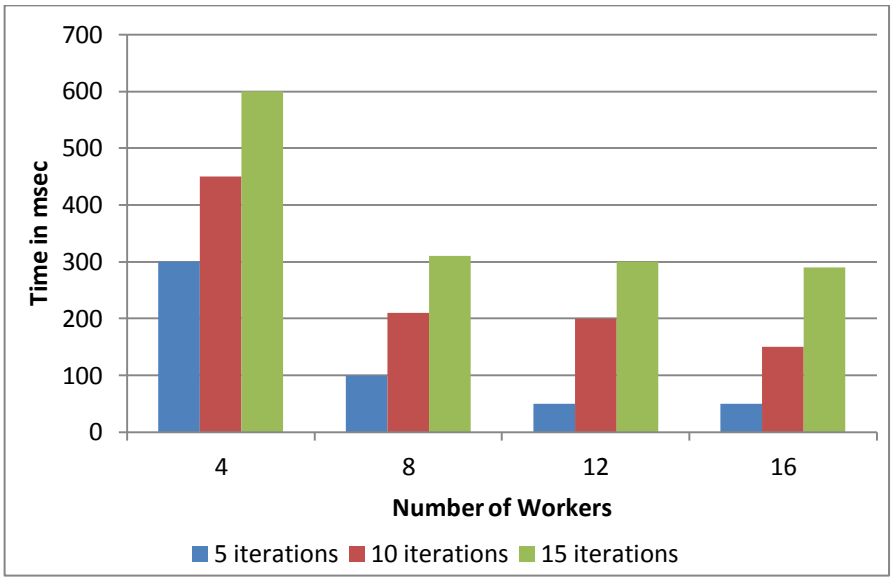
yöntemini kullanmışlardır. Reduce görevi sonlanma kriterinin sağlanıp sağlanmadığını kontrol ederek, algoritmanın devam edip etmeyeceğine karar verir.

15 iş ve 5 işlemci olduğu durumda, parçacık sürü optimizasyonu, genetik algoritma ve hibrit parçacık sürü optimizasyonu – genetik algoritma (HPSO-GA) metotlarının performanslarını Şekil 1.1'deki gibi sunmuşlardır.



Şekil 1.1 : HPSO-GA performansı [13].

Ayrıca 20 makineden oluşan bir *Hadoop* ağı üzerinde çalışan paralel HPSO-GA algoritmasının da performans değerlendirmesi aşağıda sunulmuştur. Şekil 1.2'de görüldüğü gibi ağdaki makine sayısı arttırılırken algoritmanın ölçeklenebilirliği de artmaktadır. Bunun yanında *Hadoop* sistemine gönderilen iş sayısı arttıkça iterasyon sayısı ve harcanan zaman artmaktadır.



Şekil 1.2 : Paralel HPSO-GA performansı [13].

2. GEZGİN SATICI PROBLEMİ

2.1 Problemin Tanımı

Gezgin satıcı problemi (Travelling Salesman Problem - TSP), NP-Zor bir optimizasyon problemidir [14, 22]. Bu problem, seyahat eden bir satıcının gezmesi gereken bütün şehirleri, herhangi bir şehirden başlayarak en ucuz maliyetle (en kısa yolu kullanarak) dolaşıp, tekrar başladığı şehre dönmesini vurgular. Bu tür optimizasyon problemlerinin çözümleri için günümüze kadar birçok algoritma geliştirilmiş ve çözüme ulaşılmaya çalışılmıştır. Gezilmesi gereken şehir sayısı arttıkça, problem daha da karmaşık hale gelmektedir.

Gezgin satıcı problemi, kısaca GSP, üzerinde çok durulan ve hakkında birçok araştırma yapılan tümleşik optimizasyon problemidir. Bu problem geçmiş yıllarda araştırmacıların dikkatini çekmiş ve üzerinde çalışmalar yapılmıştır. Gezgin satıcı probleminin üzerinde yapılan araştırmalar arttıkça birçok önemli araştırma alanında (olasılıklı yerel arama, tamsayı programlama) ilerlemeler kaydedilmiştir. Ayrıca gezgin satıcı probleminin üzerinde çalışılması karmaşıklık teorisinin geliştirilmesine yardımcı olmuştur. Gezgin satıcı problemi, uygulamadaki pratikliği ile yeni algoritmik fikirlerin standart test ortamı haline gelmiştir.

Çizge, uçlar ve bu uçları birbirine bağlayan kenarlardan oluşan bir tür ağ yapısıdır. GSP'nin çizge teorisindeki tanımı ise; verilen ağırlıklı bir çizgede(düğümler şehirleri, kenarlar yolları ve ağırlıklar da maliyet ya da uzaklıkları gösterir) en az maliyetli Hamilton döngüsünün bulunması olarak belirtilebilir.

GSP için simetrik [15] ve asimetrik [16] olmak üzere iki farklı tanımlama mevcuttur. Simetrik GSP yönsüz bir çizge üzerinde gösterilir ve iki düğüm arasındaki mesafe her iki farklı yönde de birbirine eşittir. Bu da olası çözüm uzayını yarıya indirger. Asimetrik GSP ise yönlü bir çizge üzerinde tanımlanabilir. Buna göre iki şehir arasındaki mesafe gidiş yönüne göre farklılık gösterebilir.

Asimetrik veya simetrik gezgin satıcı problemi matematiksel olarak tanımlanması; G=(V,E) olarak verilen ağırlıklı çizgede, i. ve j. düğümleri arasındaki ağırlığı negatif

olmayan c_{ij} değeri belirtmek üzere, tüm düğümlerden geçen minimum maliyetli turun bulunmasıdır.

Hamilton Döngüsü [17] probleminin, gezgin satıcı problemine indirgenmesi ise aşağıdaki gibi belirtilebilir:

G(V, E) verilmiş bir çizge olsun. Hamilton Döngüsü problemi, bu çizge üzerinde bütün uçlardan sadece bir kez geçip en sonunda başladığı uca ulaşan bir yol olup olmadığını bulunmasıdır. G çizgesini kullanarak kenarları ağırlıklı G^(V^, E^, w) çizgesini şu şekilde elde edilebilir: V^=V, E'deki her kenar için E^ kümesine ağırlığı 1 olan bir kenar, E'de bulunmayan her kenar içinse ağırlığı sonsuz olan bir kenar eklenir. Bu durumda, eğer G^ çizgesinde toplam ağırlığı en fazla n=|V| olan ve bütün uçlarda geçen bir yol, ancak ve ancak G çizgesinde bir Hamilton Çemberi varsa bulunabilir. Ağırlıklı G^ çizgesinde bütün uçlardan geçen ve ağırlığı en fazla n olan yol bulma problemi de gezgin satıcı problemini tanımlar.

Gezgin satıcı probleminin genellikle belirtilme biçimi aşağıdaki gibidir;

- Gezgin satıcı belirlediği şehirlerin hepsini dolaşmak istemektedir.
- Tüm şehirleri dolaştıktan sonra başladığı şehre dönmelidir.
- Bir şehirden yalnızca bir kere geçmelidir.
- Bu turu minimum maliyetle bitirmelidir.

Problemde gezgin satıcının başlangıç şehrine dönmek istemesi problemin hesaplama karmaşıklığını değiştirmemektedir.

İlk bakışta GSP modeli sadece özel birkaç uygulama için kullanışlı gibi görünmektedir. Aslında GSP, alakasız gibi görünen bir dizi uygulama alanında kullanılabilmektedir. Gezgin satıcı problemi modeli birçok alanda optimum çözümü bulmak için kullanılabilir.

- Araç Rotası Bulma: Verilen araçlarla, belli sayıda şehirlere dağılmış olan müşterilere hizmet sunmak için en iyi yolun bulunması.
- Devre Tasarımı: Belli sayıdaki pinlerin, en az sayıda çaprazlama ve kablo ile bağlanması.
- İş Zamanlama: Mevcut bir makinede çalışması gereken birbirinden farklı işlerin en etkin sırada çalıştırılması.

Gezgin Satıcı Probleminin pratikte uygulanabilir olmasından dolayı taşımacılıkta ve lojistikte kullanışlıdır. GSP'nin günümüzde klasik uygulama alanları çok fazladır, bunlardan bazıları; basılı devre imalatı, endüstride kullanılan robotların hareketlerinin zamanlanması, delme makinelerinin kuyulardaki hareketlerinin düzenlenmesi, (tek makinenin iş zamanlamasında), tatilde veya taşımacılıkta rotaların belirlenmesinde, bilgisayar ağlarında paketlerin iletiminde, gen sıralamasında, DNA dizilerinin oluşturulmasında, stok alanı malzeme toplama problemlerinde, şehirler arasında yapılacak, seyahat çizelgelemelerinde, uçaklar için havaalanı rotalamasında, telefon kulübesi, yiyecek içecek makinelerinden para toplanmasında, bankamatiklerde vs.

2.2 GSP Çözüm Yöntemleri

Herhangi bir GSP için optimum çözümü bulan tek yöntem ayrıntılı arama ve değerlendirmedir. Bu çözüm olası tüm turları bulup bunların birbirlerine göre uzunluklarını değerlendirmektir. En küçük uzunlukta olan tur en iyi turdur, bu da optimum çözümü garantilemektedir. Eğer bir turun belirlenip değerlendirilmesi bir nanosaniye sürdüğü varsayılırsa (veya saniyede bir milyar tur değerlendirildiği varsayılırsa), 25 noktadan oluşan bir GSP probleminin optimum çözümünü bulmak yaklaşık olarak on milyon yıl sürmektedir.

Açıkça daha az zamanda çözüm sunacak bir algoritmaya ihtiyaç duyulmaktadır. Daha önce de bahsedildiği gibi gezgin satıcı problemi NP-zor problem olarak belirtilmektedir, yani gezgin satıcı problemi için polinomsal zamanda çözüm üreten bilinen bir algoritma yoktur. Tam bir algoritma bulmak yerine optimum sonuca yakın veya tatmin edici sonucu kısa zamanda üreten bir algoritma bulunabilir. Gezgin satıcı problemi için birçok algoritma denenmiştir.

2.2.1 Açgözlü (Greedy) algoritma

Gezgin satıcı problemine ölçülebilir bir çözüm bulmak için geliştirilmiş yöntemdir. Algoritma öncelikle düğümlerden ve ağırlıklı kenarlardan oluşan bir çizge oluşturur ve bu çizgedeki kenarları ağırlıklarına göre sıralar. Daha sonra algoritma en kısa kenarı seçerek işlemeye devam eder; bu sırada da döngülerin oluşmasını önler. Greedy [18] algoritması makul zamanda çözüm üretmektedir ancak her zaman iyi sonuçlar verememektedir.

2.2.2 En yakın komşu algoritması

En yakın komşu algoritması [19], Greedy Algoritmasının basit yaklaşımını kullanmaktadır. Algoritma başlangıç şehrini gelişigüzel seçer ve sonra döngü oluşturmayacak şekilde en yakın şehri seçer. Algoritmanın işleyişi bu şekilde tüm şehirleri tur içine alana kadar devam eder. En yakın komşu algoritması, en son düğümün tura eklenmesi uzun sürdüğünden her zaman iyi sonuçlar verememektedir.

2.2.3 Minimum kapsayan ağaç algoritması

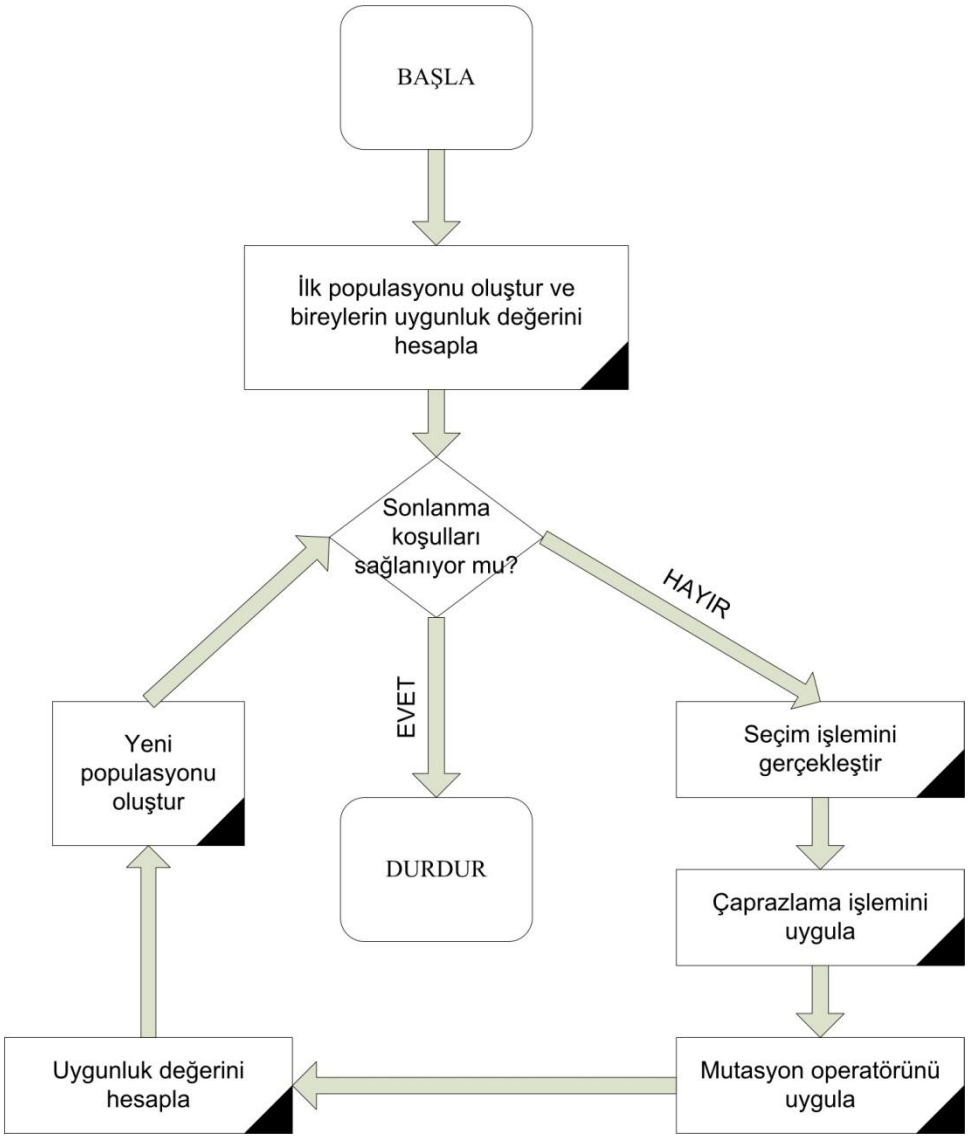
Minimum Spanning Tree [20] – En küçük kapsayan ağaç - (En Az Mesafede Tüm Düğümlere Ulaşma veya Kapsama), N şehirli çizgede N-1 kenardan geçerek tüm şehirlere ulaşmayı tanımlamaktadır, bu turda kenarların toplam uzunluğu en az olmalıdır. Çizge için bir kere minimum spanning tree bulunduğunda, düğümler arasındaki kenarlar iki yönlü olarak düşünülüp tur tamamlanabilir. Tek bir şehre bağlantısı olan herhangi bir şehirden (yaprak şehir) tura başladıktan sonra üzerinden geçilmemiş kenarlardan geçip tura devam edilebilir. Başlangıç şehrine dönene kadar bu şekilde devam edilirse gezgin satıcı probleminin çözümü için bir yöntem olabilir. Her ne kadar aynı şehirden iki defa geçilse de, bir şehirden iki defa geçmek yerine ziyaret edilmeyen şehre uğrayarak durum çözümlenebilir. Tüm şehirler gezildiği zaman doğrudan başlangıç şehrine dönülebilir.

3. GENETİK ALGORİTMALAR

Genetik Algoritmalar, genetik ve doğal seleksiyon mekanizmasına dayanan arama algoritmalarıdır [5, 24]. GA, her iterasyonda rastsal olarak daha iyi bir çözüme ulaşmaya çalışır. Bunun için de varolan çözümlere evrimsel GA operatörlerini uygular ve her iterasyon sonunda daha iyi bireylerin hayatta kalması sağlanır. Genetik algoritmanın rastsal olması çözüm uzayının farklı yönlerde taranmasına olanak tanırken, en iyilerin hayatta kalması optimum çözüme ulaşmayı sağlamaktadır.

GA bir optimizasyon problemi için aday olan çözümler popülasyonunun daha iyi sonuçlara ulaşılması amacıyla evrilmesidir. GA ilk olarak başlangıç popülasyonunun oluşturuması ile başlar. İlk popülasyon oluşturulduktan (çözüm kümesi) sonra, algoritma yinelemeli olarak; popülasyondaki bireylerin problem için uygunluk değerini hesaplar, bireyler uygunluk değerine üreme için seçilir, seçilen bireyler arasında çaprazlama işlemi uygulanarak yeni bireyler oluşturulur. Oluşturulan yeni bireyler rastsal olarak mutasyona uğradıktan sonra bir sonraki nesil için yeni popülasyon oluşturulmuş olur [1]. Genetik algoritma için akış diyagramı Şekil 3.1'de verilmiştir.

Genetik algoritma ile çalışırken, çözümün kodlanması önemli adımlardan birisidir. Üzerinde çalışılan probleme göre birçok kodlama yöntemi bulunmaktadır. İkili kodlamada çözümler bir bit dizisi olarak kodlanır. Permutasyon kodlama ise, ikili diziden farklı olarak rakamların belli bir sırada dizilmesi ile oluşturulur. Permutasyon kodlama TSP gibi sıralama problemlerinde oldukça kullanışlıdır. Değer kodlama herhangi bir tipteki veriler dizisidir ve değer olarak problemin türüne göre çeşitli tipler kullanılabilir. Ağaç kodlama, adından da anlaşılacağı gibi, çözümün ağaç yapısı üzerinde gösteriminin yapılmasına olanak sağlar [1]. Seçilen problem için hangi kodlama türünün kullanılacağı belirlendikten sonra genetik algoritmanın adımlarının gerçeklenmesine başlanabilir.



Şekil 3.1 : Genetik algoritma akış diyagramı.

3.1 Başlangıç Popülasyonunun Oluşturulması

Genetik algoritma, diğer sezgisel arama yöntemlerinden farklı olarak bir çözüm kümesi üzerinde genetik operatörler uygulayarak optimum çözümü bulmaya çalışır. GA'nın ilk adımı başlangıç kümesinin oluşturulmasıdır. İlk popülasyon rastgele üretilen kromozomlar ile ya da problem türüne göre belli bir bilgi yardımı ile sezgisel olarak oluşturulabilir. Sezgisel yöntem ile ilk popülasyonun oluşturulması optimum çözüme ulaşmayı hızlandırabilir. Ancak, bazı durumlarda çözümlerin çeşitliliğini azatabildiği için yerel optimum değerine takılma sorununa neden olur. İlk popülasyonun rastgele oluşturulması ise; çözüme daha geç ulaşılmasına neden olabilirken, çözüm çeşitliliğini arttırdığı için çözüm uzayı farklı yönlerde taranabilir ve global optimum çözüme ulaşma konusunda daha başarılıdır [21].

3.2 Amaç Fonksiyonu ve Uygunluk Değerinin Hesaplanması

Genetik algoritma, doğal evrimleşme sürecinde olduğu gibi, en iyilerin hayatta kalması prensibine göre çalışır. Buna göre bir popülasyondaki iyi bireylerin bir sonraki popülasyona aktarılma şansı, diğerlerine göre daha yüksektir. Bu noktada önemli olan hangi bireylerin diğerlerine göre daha iyi olduğunun tespit edilmesidir. Bunun için amaç fonkisyonu kullanılmaktadır. Amaç fonksiyonu bireylerin ne kadar iyi olduklarına göre o bireylere bir uygunluk değeri atar. Amaç fonksiyonunun belirlenmesi çözüm aranan probleme göre değişiklik gösterir. Örneğin, bir maksimizasyon probleminde verilen ifadeyi maksimuma taşıyan bireyler daha yüksek uygunluk değerine sahip olurken, gezgin satıcı problemi için ise, bireyin temsil ettiği turun maliyeti ne kadar az ise bireyler o kadar yüksek uygunluk değerine sahip olacaklardır.

3.3 Uygun Bireylerin Seçimi

Popülasyondaki bireylerin bir kısmı sonraki kuşaklara aktarılırken, bir kısmı da yok olmak zorunda kalır. Bu aşamada hangi bireylerin sonraki kuşaklara aktarılacağı kullanılan seçim metodu ile belirlenir. Bireylerin seçimi sahip oldukları uygunluk değerine göre yapılır. En iyi bireyler yeni kuşak üzerinde daha çok etkiye sahipken, düşük uygunluk değerine sahip bireylerin sonraki kuşak üzerinde hiç etkisi olmayabilir. Hayatta kalma mücadelesi olarak adlandırılan bu durum, GA'nın optimum sonuca ulaşmasında önemli adımlardan birisidir.

3.3.1 Rulet tekerleği seçimi

Jenerasyon sayısı ilerledikçe, bireylerin gen yapılarının belirli bir şema tarafından domine edilmesinin önüne geçilmelidir. Aksi takdirde, ilerleyen iterasyonlarda tüm bireylerin benzer bir çözüm sunması ve algoritmanın yerel bir optimum değere takılı kalması kaçınılmazdır. Rulet tekerleği seçiminde bireyler uygunluk değerlerine göre bir rulet tekerleğinde belli bir yüzde ile alan kaplarlar. Dolayısı ile tekerleğin döndürülmesi ile bir bireyin sonraki nesillere aktarılma olasılığı tekerlek üzerinde kapladığı alan ile doğru orantılı olarak değişir. Bu yöntem düşük uygunluk değerine sahip bireylere seçilme şansı tanırken, ortalama uygunluk değerinden çok daha yüksek uygunluk değerine sahip bir bireyin tüm popülasyon üzerinde egemen olmasına yol açar [1].

3.3.2 Rank seçimi

Rulet tekerleği seçimi, uygunluk değerlerinin çok fazla değişken olduğu bir durumda problem yaratacaktır. Bir kromozomum uygunluk değeri %90'lık bir alanı kaplarsa, rulet tekerleğinde diğer bireyler için çok küçük bir alan kalacak bu da tek bireyin gelecek kuşakta egemen olmasına neden olacaktır. Bunu önlemek için rank seçimi kullanılmaktadır. Rank seçiminde bireyler uygunluk değerine göre sıralanırlar. En kötü birey 1, en kötü ikinci birey 2 değerlerini alacak şekilde rank ataması yapılacaktır. Popülasyon büyüklüğünün N olduğu bir durumda en iyi birey N değerini alacaktır. Bireyler rank değerlerine göre rulet tekerleği üzerinde alan kaplayacaklardır. Böylece hiçbir zaman tek birey tüm popülasyona egemenlik kuramayacaktır ve çözüm uzayının arama yönünün değiştirilmesi sağlanacaktır [1].

3.3.3 Turnuva seçimi

Turnuva seçim yöntemi popülasyondan belirlenen sayıdaki bireyi rastgele alır ve aralarından en iyi olan bireyi seçer. En iyi birey en yüksek uygunluk değerine sahip bireydir. Yeterli sayıda birey seçimi gerçekleşene kadar turnuva devam eder. Her turnuvanın galibi çaprazlama için seçilmiş olur. Eğer turnuvaya katılacak birey sayısı çok fazla olursa uygunluk değeri düşük olan bireylerin seçilme şansı azalacaktır.

3.3.4 Seçkincilik

Bireyler üzerinde çaprazma ve mutasyon operatörleri uygulanırken, popülasyonun sahip olduğu en iyi birey (çözüm) kaybolabilir. Bunu önlemek için popülasyondaki en iyi birey ya da belirlenen sayıdaki en iyi bireylerin doğrudan bir sonraki kuşağa aktarılması işlemine seçkincilik ismi verilir. Seçkincilik GA'nın performansını arttırıcı bir etkiye sahiptir. Böylece bir sonraki kuşaktaki en iyi bireyin bir önceki kuşaktaki en iyi bireyden daha kötü olma ihtimali kalmaz. Ancak yeni nesile aktarılacak bireylerin sayısı iyi ayarlanmalıdır. Yeni nesile aktarılacak en iyi bireylerin sayısı çok fazla olduğunda algoritma hep aynı çözüm kümesinin etrafında dolaşacak ve global optimuma erişme olasılığı azalacaktır [1].

3.4 Çaprazlama

Genetik algoritma her jenerayonda varolan çözümleri daha iyiye götürmek için çaprazlama operatörünü kullanır [25, 26]. Seçim aşamasında anlatıldığı üzere

çaprazlama, seçilen iki veya daha fazla bireyin genlerinin kullanılarak yeni bireylerin elde edilmesi işlemini gerçekleştirir.

Çaprazlama işleminde önemli olan bir konu da, çaprazlama noktasının çaprazlamadan elde edilecek çocuk kromozomların uygunluk değerleri üzerindeki etkisidir. Bu işlem yapılırken her zaman sonuçlar önceden tahmin edilemez. Bu yüzden gelişigüzel yapılan değişikliklerde sonucun mükemmelliğe doğru gitmesi için belirli kriterler bulmak için çalışılır. Kromozomlardaki genlerin yapısı ve etkileri araştırılarak, bu genlere yapılan müdahalelerle bireye bazı iyi özellikler kazandırılabilir. Çaprazlamadan elde edilecek çocuk kromozomların uygunluk değeri bir önceki ana kromozomlardan daha yüksek olmayabilir.

Çaprazlama operatörü genetik algoritmaların üzerinde en fazla çalışma yapılan alanıdır. Her problem için farklı bir çaprazlama yöntemi kullanılması gerektiğinden günümüze kadar çok çeşitli çaprazlama yöntemleri ortaya atılmıştır.

3.4.1 Tek noktalı çaprazlama operatörü

Tek noktalı çaprazlama yönteminde krozomon üzerinde 1 ile L (kromozom uzunluğu) arasında rastgele bir C noktası seçilir. Ve bu nokta seçilen iki birey için çaprazlama noktası olarak kabul edilir. İki birey bu noktanın sonrasındaki kısımlarını aralarında değiştirerek iki tane yeni birey oluşturulur.

3.4.2 İki noktalı çaprazlama operatörü

İki noktalı çaprazlama yönteminde çaprazlama için 1 ile L (kromozom uzunluğu) arasında C1 ve C2 olmak üzere iki adet nokta seçilir. Seçilen iki bireyin bu iki nokta arasındaki genlerinin birbiri ile değiştirilmesi ile iki adet yeni birey elde edilir.

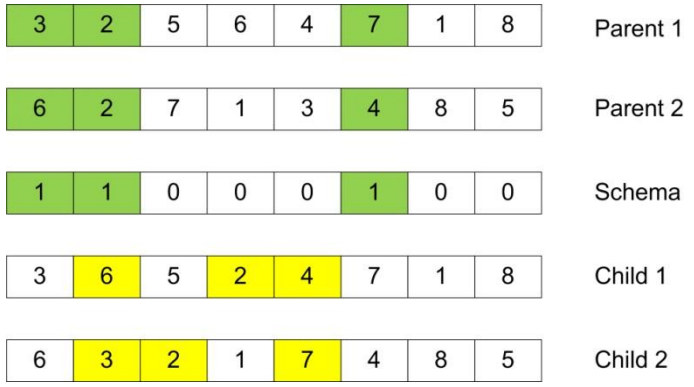
3.4.3 Pozisyona dayalı çaprazlama operatörü

Pozisyona dayalı çaprazlama yönteminde [27] hangi pozisyondaki genlerin seçilen iki birey arasında değiştirileceği, Şekil 3.2‘de gösterildiği gibi, önceden belirlenen bir kalıp doğrultusunda yapılır. Buna göre değişimin olmayacağı genler çocuk kromozoma doğrudan aktılırken, değişimin gerçekleştirileceği genler diğer kromozomdan alınır.

Şekil 3.2 : Pozisyona dayalı çaprazlama.

3.4.4 Sıraya dayalı çaprazlama yöntemi

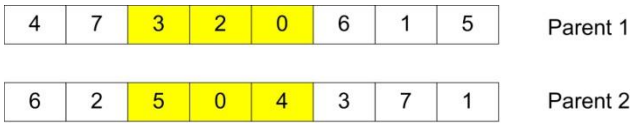
Şekil 3.3'de gösterilen sıraya dayalı çaprazlama yönteminde hangi pozisyondaki genlerin bireyler arasında değiştirileceği önceden belirlenen kalıp doğrultusunda yapılır [27]. Diğer bireydeki genler şemada belirtilen sırayla alınır (6, 2, 4). Daha sonra bu genlerin kopyalanacak olan bireydeki konumları tespit edilir (2. 4. ve 5. genler). Belirlenen genlerin haricindeki genler çocuk bireye aynen kopyalanırken, değişecek genlerin yerine diğer bireyde belirlenen genler sırasıyla eklenir.



Şekil 3.3 : Sıraya dayalı çaprazlama.

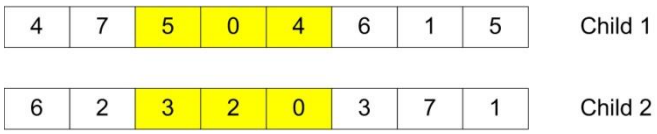
3.4.5 Kısmi planlı çaprazlama yöntemi

Kısmi planlı çaprazlama yöntemi iki noktalı çaprazlama yöntemini kullanır. Bu yöntemde seçilen iki nokta arasındaki genler bir ebeveynden alınırken, kalan genler sırası ile diğer ebeveynden alınır. Eğer ikinci ebeveynden alınan genler zaten oluşturulan bireyde var ise, onun yerine kullanılacak olan gen oluşturulan bir eşleme tablosundan alınır [26]. Böylece GSP için uygun bir çaprazlama operatörü olarak kullanılabilir. Çaprazlama için seçilen iki kromozomun Şekil 3.4'deki gibi olduğu bir durumda; öncelikle iki adet çaprazlama noktası seçilir.



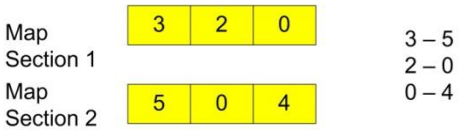
Şekil 3.4 : Çaprazlama için seçilen bireyler.

Bu iki nokta arasındaki genler ebeveynler arasında yer değiştirilerek Şekil 3.5'de gösterilen prototip yeni bireyler oluşturulur. Ayrıca bu iki nokta arasındaki genlerin eşlemesi yapılır.



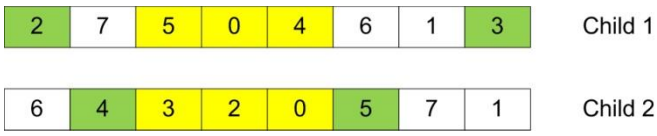
Şekil 3.5 : Prototip bireyler.

Görüldüğü üzere prototip bireyler aynı genleri tekrar etmektedir. Bu durumu çözmek için oluşturulan Şekil 3.6'daki eşleme tablosu kullanılır.



Şekil 3.6 : Eşleşme tablosu.

Eşleşme tablosuna göre prototip bireylerin tekrar eden genleri düzeltilir. Böylece Şekil 3.7'de görüldüğü gibi iki yeni birey elde edilmiş olur.



Şekil 3.7 : Oluşturulan yeni bireyler.

3.4.6 Açgözlü çaprazlama

Açgüzlü çaprazlamanınn kullanılabilmesi için; bireyin tüm genlerinin alacağı değerlerin farklı olması ve çaprazlama yapılacak olan bireylerin genlerinin sırası farketmeksizin aynı olması gerekmektedir. Bu durum GSP problemi için uygun bir örnektir.

Algoritma öncelikle iki bireyden birisinden rastgele bir başlangıç noktası seçer. Daha sonra her iki ebeveyndeki bu noktaya bağlı diğer nokta bulunur. Bu iki noktanın seçilen noktaya maliyetleri karşılaştırılır ve maliyeti az olan tura eklenir. Eğer noktalardan herhangi bir tanesi daha önce turda mevcut ise diğer nokta tura eklenir.

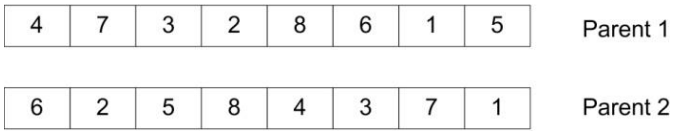
Eğer her iki şehir de tura daha önceden eklenmiş ise; eklenmemiş şehirlerarasından rastgele bir tanesi tura eklenir. Ve tur tamamlanıncaya kadar aynı prosedür devam eder [2, 28].

Maliyet tablosunun Çizelge 3.1'deki gibi verildiği bir problem için;

Çizelge 3.1 : Komşuluk matrisi.

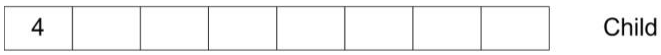
x	50	40	70	80	20	10	70
	x	30	20	60	50	20	40
		x	40	30	60	30	10
			x	60	30	10	50
				x	20	60	20
					x	40	40
						x	30
							x

Çaprazlama için seçilen kromozomların (P1, P2), Şekil 3.8'deki gibi olduğu varsayılsın.



Şekil 3.8 : Çaprazlama için seçilen ebveyn bireyler.

Öncelikle herhangi bir şehir başlangıç şehri olarak seçilir. Şekil 3.9'da gösterilen 4 numaralı şehir başlangıç şehri olarak seçilmiş olsun.



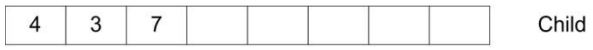
Şekil 3.9 : Başlangıçtaki çocuk birey.

Daha sonra her iki ebeynde 4'ten sonraki şehirlerden en az maliyete sahip olan ve daha önce çocuk kromozoma aktarılmamış olan şehir seçilir. Eğer ebeynlerden herhangi birindeki şehir, çocuk kromozomda mevcut ise diğer kromozomdaki bir sonraki şehir, çocuk kromozoma eklenir. Eğer her ikisinde de bir sonraki şehir, çocuk kromozomda mevcut ise; daha önce seçilmemiş olan şehirlerden en son eklenen şehre (4) en yakın olan şehir, çocuk kromozoma eklenir. Her iki ebeveyndeki bir sonraki şehirlerin maliyeti eşit ise bir sonraki şehir ebeveynlerden herhangi birisinden rastgele seçilir.

Bakıldığında P1 için 4'ten sonraki şehir 7, P2 için ise 3'tür. 4-3 yolunun maliyetinin 4-7'ye eşit olmasından dolayı rastgele seçimle, 3 bir sonraki şehir olarak seçilmiş olsun. Yeni bireyin son hali Şekil 3.10'daki gibi olacaktır.

Şekil 3.10 : İkinci ebeveyndeki şehrin eklenmesi sonucu oluşan çocuk birey.

Yukarıda anlatılan işlem tekrarlanır. Bu durumda P1 için 3'ten sonraki şehir 2, P2 için 3'ten sonraki şehir 7'dir. 3-7 yolunun maliyeti daha küçük olduğu için 7 bir sonraki şehir olarak çocuk kromozoma eklenir (Şekil 3.11).

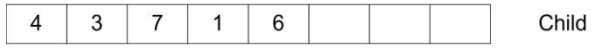


Şekil 3.11 : 3-7 yolunun kısa olması nedeni ile 7 numaralı şehir çözüme eklenir.

Sonraki karşılaştırma için 7-3 ve 7-1 seçilir. 3 daha önce kromozoma eklendiği için, 1 çocuk kromozoma eklenir (Şekil 3.12).

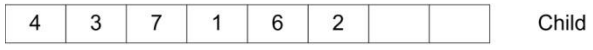
Şekil 3.12 : Döngüye yol açmayan 1 numaralı şehir listeye eklenir.

P1 için 1-5, P2 için 1-6 bir sonraki seçimde kullanılır. 1-6 daha küçük maliyete sahip olduğu için 6 sonraki şehir olarak kromozoma eklenir (Şekil 3.13).



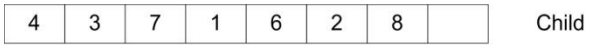
Şekil 3.13 : 1-6 yolunun maliyetinin daha az olması nedeniyle 6 çözüme eklenir.

6-1 ve 6-2 bir sonraki seçimde kullanılacaktır. Ancak 1 daha önce eklendiği için 2 çocuk kromozoma eklenir (Şekil 3.14).



Şekil 3.14 : Döngüye yol açmadığı için 2 numaralı şehir çözüme eklenir.

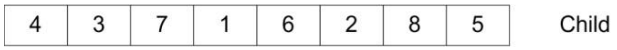
2-8 ve 2-5 yollarından 2-8 daha küçük maliyete sahip olduğu için 8 çocuk kromozoma eklenir (Şekil 3.15).



Şekil 3.15 : Maliyetinin daha az olması nedeni ile 8 listeye eklenir.

8-6 ve 8-4 yollarının her ikisi de daha önce eklendiği için 8'e komşu ve daha önce eklenmemiş en yakın şehir seçilir. Bu durumda sadece 5 seçilmemiş şehir olarak

kaldığı için 5 çocuk kromozoma eklenir ve Şekil 3.16'daki iki farklı ebeveynden farklı bir yeni birey oluşturulmuş olur.



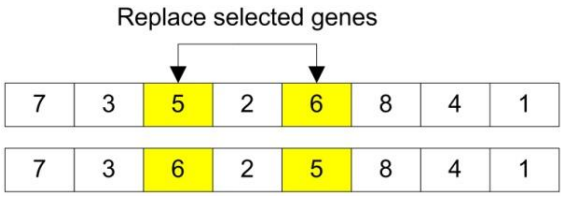
Şekil 3.16 : Oluşturulan yeni birey.

Sonuçta iki ebeveynin en iyi genlerini kullanan ve iki ebeynden farklı bir yeni birey oluşturulur. Açgözlü çaprazlama ile her jenerasyonda yerel optimum değerine ulaşılmaya çalışılırken, mutasyon ile çözüm uzayının farklı yönlerde taranması gerçekleştirilir.

3.5 Mutasyon

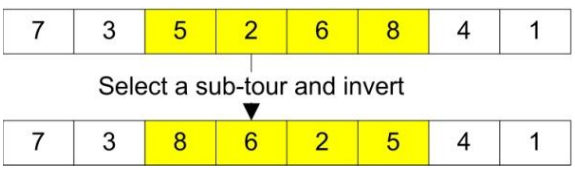
GA'da sistem belli döngü değerine geldikten sonra bireyler birbirlerine daha da benzemektedir. Bu da çözüm uzayınında daha küçük bir alanın taranmaya başlamasına neden olmaktadır. Bireyler üzerinde ne kadar çaprazlama operatörü uygulansa da çeşitlilik zamanla kaybolmaktadır. Bu durumda bireyin kendi içinde gen değerleri rasgele yer değiştirilir. Böylelikle popülasyondaki bireylerin çeşitliliğinin devamı sağlanmış olur [28]. Mutasyon için birçok çeşit bulunmaktadır ve bunlardan bazıları aşağıda açıklanmıştır.

En basit mutasyon yöntemi olan yer değiştirme yönteminde, Şekil 3.17'de görüldüğü gibi seçilen rastgele iki genin yeri değiştirilir.



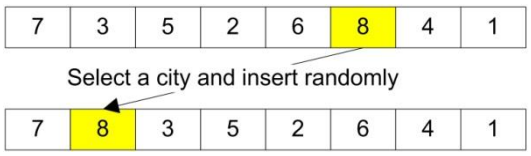
Şekil 3.17 : Yer değiştirme yöntemi.

Ters çevirme yönteminde, kromozom içersinde bir alt-tur seçilir ve bu tur ters çevrilerek seçilen alt-turun yerine konulur (Şekil 3.18).



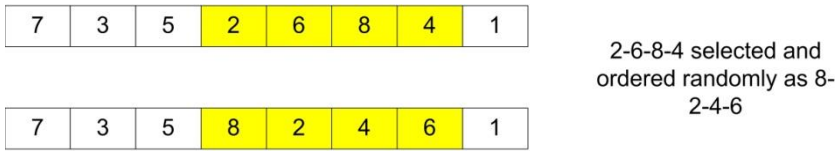
Şekil 3.18 : Ters çevirme yöntemi.

Ekleme mutasyon metodunda, Şekil 3.19'da gösterildiği gibi seçilen herhangi bir şehir, yine kromozomun herhangi bir noktasına ekler.



Şekil 3.19 : Ekleme yöntemi.

Şekil 3.20'de görülen karıştırma (scramble) metodunda rastgele bir alt-tur seçilir. Ve alt tur kendi içersinde rastgele sıralanarak tekrar kromozoma eklenir.



Şekil 3.20 : Karıştırma yöntemi.

Mutasyon operatörünün uygulanma oranı doğru belirlenmelidir. Mutasyon oranının yüksek olması çözüm uzayını çok genişleterek sistemin yanlış yerlerde arama yapmasına neden olur. Mutasyon oranının çok küçük belirlenmesi ise global optimum değerine ulaşmayı engelleyecektir. Mutasyon oranı, üzerinde çalışılan probleme ve geliştirilen GA'nın kullandığı yönteme göre deneysel olarak tespit edilir.

3.6 Genetik Algoritmada Parametrelerin Seçimi

Genetik algoritmalarda parametre seçimi çözümün kalitesini ve çözüm hızını etkileyen en büyük faktördür [5]. Parametre seçimi için her ne kadar genel kurallar var ise de, parametre seçimi ele alınan probleme bağlı olarak değişmektedir. Örneğin, bazı GA gerçekleştirimlerinde mutasyon oranı çok küçük seçilirken, bazı uygulamalarda çaprazlama operatörü kullanılmaz ve sadece mutasyon kullanılarak bireylerin gelişimi sağlanabilir. Popülasyon büyüklüğü, çaprazlama ve mutasyon olasılığı, seçim yöntemi GA'nın en bilinen parametreleridir [5].

3.6.1 Popülasyon büyüklüğü

Popülasyon büyüklüğünün ayarlanması önemli aşamalardan biridir. Buna bağlı olarak, eğer popülasyon çok büyük olursa optimum çözüme ulaşma zamanı uzayacaktır. Bu değerin çok küçük olması durumunda ise, algoritma yerel optimuma

takılabilir. Bunun için dikkat edilen nokta problem büyüklüğüdür. Yani kromozomdaki gen sayısı arttıkça popülasyon büyüklüğünün artması daha iyi sonuçlar verecektir.

3.6.2 Çaprazlama ve mutasyon olasılığı

Çaprazlamada amaç bireylerin aralarında çözümün belli parçalarını değiştirerek daha iyi sonuçların üretilmesini sağlamaktır. Çaprazlama oranının yüksek tutulması daha iyi bireylerin oluşma şansını arttırırken, aynı zamanda popülasyondaki en iyi bireyin kaybolmasına da yol açabilir. Popülasyondaki en iyi bireyin kaybolmasını engellemek için elitizm yöntemi [1] kullanılarak o anda popülasyondaki en iyi N bireyin bir sonraki nesile aktarılması sağlanabilir.

Genetik algoritmada nesil sayısı ilerledikçe bireylerin birbirine benzerlikleri artmakta ve aranan çözüm uzayı küçülmektedir. Bu durum algoritmanın yerel optimum değerine saplanmasına sebep olabilmektedir. Bu durumun engellenmesi için belirli bir olasılıkla bireyler üzerinde mutasyon işlemi uygulanır. Mutasyon olasılığının yüksek olması algoritmanın optimum çözüme ulaşmasını geciktirirken, mutasyon olasılığı çok küçük olduğu durumlarda algoritmanın yerel optimum değere takılması durumu ortaya çıkmaktadır.

3.6.3 Seçim yöntemi

Yeni nesil oluşturulurken seçilen bireyler arasında çözümün belli parçalarının değiştirilmesiyle yeni bireylerin oluşması sağlanır. Çaprazlama işlemi uygulanacak bireylerin belirlenmesi için çeşitli çözüm yöntemleri bulunmaktadır. Uygunluk değeri en iyi olan bireylerin çaprazlama için seçilmesi durumunda algoritma daha hızlı bir şekilde sonuca ulaşmaktadır. Ancak bu durumda nesiller ilerledikçe bireylerin tek düze hale gelmesi söz konusu olmaktadır. Diğer bir seçim yöntemi ise, sürekli en iyi bireylerin seçilmesi yerine bazen kötü bir bireyinde çaprazlama için seçilmesidir. Bu durum algoritmanın çözüm uzayını farklı yönlerde taramasına olanak sağlamaktadır. Ancak algoritma daha yavaş şekilde evrilecektir [1].

Seçim yöntemlerinde dikkat edilmesi gereken diğer bir konu ise, çaprazlama için seçilen bireylerin tekrar seçilip seçilemeyeceğine karar vermektir. Seçilen bireylerin tekrar seçim havuzuna konulması bir sonraki seçimde iyi bireylerin tekrar seçilme şansını arttırırken, ilerleyen nesillerde çözümlerin benzer hale gelmesine yol açar.

Bu çalışmada ise doğal yaşamda “akraba evliliği” olarak adlandırılan durum engellenmiştir. Bu bağlamda, çaprazlama için seçilecek olan bireylerin aralarındaki gen benzerlik oranı belirli bir yüzdeyi geçemeyecektir. Böylece herhangi bir çözümün belirli bölgesinde oluşan hatalı bir çözüm parçasının bir sonraki nesle aktarılmasının önüne geçilmiş olur. Benzerlik oranı çok küçük seçildiğinde, algoritma çok farklı iki bireyi çaprazlama işlemine tabi tutar. Bu durumda optimum çözüme ulaşmak güçleşir. Benzerlik oranının çok yüksek tutulması durumunda ise; önerilen yöntem GA performansına herhangi bir etki yapmayacaktır [2].

4. PARALEL GENETİK ALGORİTMALAR

4.1 Giriş

Daha önce de bahsedildiği gibi, GA, uygunluk değeri hesaplama, seçme, çoğalma ve mutasyon aşamalarından oluşmaktadır. GA'nın en çok zaman harcayan kısmı uygunluk değerinin hesaplanması aşamasıdır [6]. Özellikle popülasyon büyüklüğünün çok fazla olduğu problemlerde bireylerin uygunluk değerinin hesaplanması çözülmesi zor bir problem haline gelmektedir. GA problemlerinden bir diğeri ise algoritmanın zaman zaman yerel optimum değerine takılmasıdır. Ardışıl genetik algoritmanın (Sequential Genetic Algorithm - SGA) bazı durumlarda yetersiz kalması onların paralelleştirilmesi fikrini ortaya çıkarmıştır.

SGA'nın yetersiz kaldığı ve paralel genetik algoritmalar (PGA) ile nasıl çözümlenebileceği aşağıda belirtilmiştir:

- Bazı problemlerin çözümü için popülasyon sayısının çok fazla olması gerekmektedir. Bu durumda popülasyondaki tüm bireyleri saklamak için çok fazla bellek ihtiyacı doğmaktadır. Paralel genetik algoritmalar ile bu sorun çözülebilir [6].
- Uygunluk değerinin hesaplanması en çok zaman gerektiren kısımdır. Bazı durumlarda 1 işlemci yılı zaman gerektirdiği raporlanmıştır. Uygunluk değeri hesabının paralel şekilde yapılması süreyi kayda değer şekilde azaltacaktır [6].
- Ardışıl genetik algoritmalar bazen yerel optimum çözüme takılı kalabilmektedir. Paralel genetik algoritmalarda aynı anda birden fazla alt-popülasyon çözüm uzayını farklı yönlerde tarayabildiği için global optimuma ulaşma olasılığı çok daha yüksektir [6].

Paralel genetik algoritmalar tek bir makinede çalıştırılsa bile; birden fazla alt-popülasyon içerdiği ve bunlar da farklı yönlerde geliştiği için ardışıl çözüme göre daha iyi sonuç verecektir.

Paralel genetik algoritmalar bir kaç gruba ayrılabilir. Gruplama işlemi ise bazı kriterlere göre yapılır. Bu kriterler aşağıdaki gibi özetlenebilir:

- Uygunluk değerinin hesaplanma ve mutasyon operasyonunun uygunlanma şekline göre,
- Tek ya da birden fazla popülasyon olmasına göre,
- Birden fazla popülasyon varsa popülasyonlar arasındaki birey değişiminin şekline göre,
- Seçme işleminin global ya da yerel uygulanmasına göre.

4.2 Master – Slave Paralelleştirme

Bu gruptaki algoritmalarda tek popülasyon vardır. Popülasyondaki bireylerin uygunluk değeri hesabı paralel şekilde gerçekleştirilirken, diğer genetik operatörler ardışıl olarak uygulanır. Senkron ve Asenkron olmak üzere iki şekilde gerçeklenebilir. Senkron gerçelemede, tüm bireylerin uygunluk değeri hesabının yapılmış olması beklenir. Daha sonra genetik algoritmanın diğer adımlarına geçilir. Bu durumda algoritmanın çalışma hızını ağdaki en yavaş makine belirleyecektir. Asenkron gerçeklemede ise, belli sayıda birey için uygunluk değeri hesaplandığında, o bireyler için bir yarışma seçimi uygulanır ve seçilen bireyler için algoritma çalışmaya devam eder.

4.3 Göç Alan Statik Alt-popülasyonlar

Popülasyon birçok alt-popülasyondan oluşur. Birbirinden tamamen bağımsız olan alt-popülasyonlar arasında bireylerin göç etmesi sağlanır. Bu işlem için ayrıca göç operatörü tanımlanır. En popüler paralelleştirme yöntemidir. Eğer bireyler herhangi bir alt-popülasyona göç edebiliyorlarsa buna “ada metodu” adı verilir. Bireyler sadece komşu alt-popülasyonlara göç edebiliyorsa buna “atlama tahtası metodu” adı verilir [6]. Bireylerin göç etmesini kontrol eden bazı parametreler aşağıdaki gibidir:

- Alt-popülasyonlar arasındaki fiziksel ağ bağlantısı(küp, iki/üç boyutlu örgü)
- Göç oranı (Kaç adet birey göç edecek)
- Göç edecek bireyler (En iyi bireyler, rastgele)
- Değiştirilecek bireyler (En kötü, rastgele)
- Göç aralığı (göç olma sıklığı)

4.3.1 İri taneli algoritmalar

- Az sayıda popülasyon
- Her popülasyonda çok fazla sayıda birey

4.3.2 İnce taneli algoritmalar

- Çok sayıda alt-popülasyon (çok fazla sayıda işlemci gerektirir)
- Makine sayısı arttıkça alt-popülasyonlardaki birey sayısını ayarlamak zorlaşır

Bu algoritmalar ardışıl genetik algoritmalar gibi çalışır. Sadece göç operatörü tanımlanması yeterli olacaktır.

4.4 Statik Kesişen Alt-popülasyonlar

Birden fazla alt-popülasyon içerir. Alt-popülasyonların belli bölgelerinde kesişimler oluşturulur ve bu noktalardaki bireyler her iki alt-popülasyonda da bulunmaktadır. Bu bireyler birden fazla seçim ve çaprazlama işlemine tabi tutulabilir. Kesişim noktalarındaki bireyler yardımı ile alt-popülasyonlar arasında iletişim/değişim sağlanmış olur. Bu gruptaki algoritmalar paylaşılan-bellek mimarileri için uygundur.

4.5 Yoğun Paralel Genetik Algoritmalar

Kesişen statik alt-popülasyonlarda, alt-popülasyon sayısı arttırılıp, popülasyondaki birey sayısı azaltıldığında (ince-taneli) yoğun paralel genetik algoritmalar elde edilmiş olur. Yoğun paralel mimariler için uygundur. Her işlemciye bir birey düşer ve mimariye bağlı olarak (2 boyutlu, 16x8x8 küp) bireyler arasındaki komşuluk ilişkileri düzenlenir. Bu tür genetik algoritmalar çok işlemcili makinelerde simüle edilebilir. Çalışma istasyonlarında da gerçekleştirim yapılabilir ancak bu durumda iletişim maliyeti çok yüksek olacaktır.

4.6 Dinamik Alt-popülasyonlar

İri-taneli algoritmalar ile asenkron master-slave algoritmaların bir birleşimidir. Herhangi bir göç operatörüne ihtiyaç yoktur. Çoklu komut çoklu veri mimarisi için uygundur. Master-slave paralelleştirimindeki en yavaş işlemciyi bekleme sorununu çözer. Popülasyon dinamik olarak alt-popülasyonlara bölünür. Paylaşımlı veya dağıtık bellek mimarileri üzerinde çalışabilir [6].

5. HADOOP – MAPREDUCE MİMARİSİ

Hadoop büyük veri dosyalarının paralel şekilde işlenmesi için kullanılan *MapReduce* yönteminin açık kaynak kodlu gerçekleştirimidir [9]. Bilgisayar kümeleri üzerinde çalışan dağıtık programlar için alt yapı desteği sunar. Kullanıcı etkileşimi gerekmeksizin verilerin ilgili düğüme taşınmasını ve olası bir donanım sorununda kurtarma işlemlerini otomatik gerçekleştirir. Kullanılan *MapReduce* yönteminde, yazılan program bilgisayar kümesi üzerinde bulunan bir düğümde çalışabilecek küçük parçalara ayrılır. *Hadoop* ayrıca bilgisayar kümesi için yüksek bant genişliği sağlayan dağıtık dosya sistemini (Hadoop Distributed File System - HDFS) kullanıcıya sunar [7]. Hem *MapReduce* hem de HDFS dosya sisteminin kullanımı ile birlikte düğüm hataları otomatik olarak düzeltilir ve kullanıcıya yüksek güvenilirlikli bir sistem sunulur.

5.1 MapReduce Mimarisi

MapReduce, Hadoop yapısının bir işi bilgisayar kümesi üzerine dağıtmasını sağlayan anahtar algoritmadır. *MapReduce* iki ayrı safhadan oluşur: map safhası ve reduce safhası. Her iki safha da girdi ve çıktı olarak anahtar/değer çifti gerektirir. Kullanılacak olan anahtar/değer çifti kullanıcı tarafından belirlenir. Kullanıcı ayrıca map ve reduce fonksiyonlarını tanımlamalıdır.

5.1.1 Map fonksiyonu

Map fonksiyonunun amacı girdi olarak gelen verilerin gruplanıp, sıralanıp çıktılarının reduce fonksiyonuna aktarılmasıdır [9]. *Hadoop* yapısında, *MapReduce* işine girdi olarak verilen dosya HDFS dosya sisteminde olmalıdır. Dosyanın her satırı anahtar/değer şeklinde bir kayıt olarak okunur. Anahtar değeri okuma tipine göre byte cinsinden satırın başlangıç adresi, değer ise satırın metin kısmıdır. Okunan kayıtlar HDFS dosya sisteminin blok boyutuna göre gruplanır. Ve gruplanan bu kayıtlara, map görevinin girdisi olan, "split" adı verilir.

Şekil 5.1'deki örnekte Map fonksiyonu girdi anahtar/değer olarak long/text tipinde değer almaktadır. Örnekte anahtar değeri ile ilgili bir işlem yapılmamaktadır. Map

fonksiyonu değer olarak ise satırı alır. Satırı kelimelerine ayırıp çıktı olarak ayrıştırılan kelime ve kelimenin sayısını çıktıya aktarır. Çıktı anahtar/değer ikilisi olarak text/int (bulunan kelime/kelimenin sayısı) dışarıya verilir.

```
package my.pack;

import java.io.IOException;
import java.util.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException
    {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
         while (tokenizer.hasMoreTokens())
         {
             word.set(tokenizer.nextToken());
             context.write(word, one);
         }
    }
}
```

Şekil 5.1 : Map fonksiyonu.

5.1.2 Reduce fonksiyonu

Reduce fonksiyonu bilgisayar kümesinde çalışan map fonksiyonlarının çıktılarını girdi olarak alır. Dolayısı ile map fonksiyonunun çıktı anahtar/değer ikilisi ile reduce fonksiyonunun girdi anahtar/değer ikilisinin tipleri aynı olmalıdır [9]. Reduce fonksiyonu anahtar düzeyinde sıralanmış ve gruplanmış girdi değerlerini alır ve çözüm için ilgili işlemleri gerçekleştirir.

Reduce fonksiyonu, Şekil 5.2'de gösterildiği üzere girdi anahtar/değer ikilisi olarak text/int(dizi) almaktadır. Bu ikili map fonksiyonunun çıktı tipi ile aynıdır. Map fonksiyonununda çıktı değeri olarak sadece int kullanılırken reduce fonksiyonu değer olarak int tipinde bir dizi almaktadır. Çünkü aynı anahtar değerine sahip çok fazla değer olabilir. Reduce fonksiyonu girdi olarak aldığı anahtarı (ilgili kelime) direk çıktı olarak yazar. Girdi olarak aldığı değer parametresini ise "sum" değişkenine ekler. Burada amaç aynı anahtar değerine sahip olan değerlerin sayısını bulmaktır.

```
package my.pack;

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class MyReducer extends Reducer <Text, IntWritable, Text, IntWritable>
{
    public void reduce (Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException
    {
        int sum =0;
        for (IntWritable value : values)
        {
            sum = sum +value.get();
        }

        context.write(key, new IntWritable(sum));
    }
}
```

Şekil 5.2 : Reduce fonksiyonu.

Hadoop, MapReduce işine girdi olarak verilen dosyayı "split" adı verilen sabit-büyüklükteki parçalara ayırır [10]. Daha sonra her "split" için bir map fonksiyonu çalıştırılır. Map fonksiyonu "split" içerisindeki kayıtları (anahtar/değer girdileri) işler. Split boyutu HDFS blok boyutuna eşit olmalıdır. Blok boyuttan daha küçük olduğunda map görevlerinin bir defada okuyabileceği veriyi daha fazla tekrarda okumak zorunda kalacaktır. Split boyutunun HDFS blok boyutundan büyük olduğu durumda map fonksiyonuna giriş olarak verilen "split"in bir kısmı başka bir düğümde yer alabilir ki, bu da veri lokalizasyonuna aykırı olacaktır. Yani map görevi girdi değerini başka bir düğümden okumak için ağ üzerinden okuma yapmak zorunda kalacaktır [7].

Reduce görevleri ise "veri lokalizasyonu optimizasyon"undan yararlanamaz, çünkü reduce görevi girdi olarak birçok map görevinin çıktılarını kullanmaktadır. Dolayısı ile tüm map görevleri aynı düğümde olmayacağı için reduce görevleri ağ üzerinden okuma yapmak zorunda kalcaktır [7].

Ağ üzerinden veri aktarım miktarını azaltmak için map ve reduce görevlerinin arasına bir birleştirici fonksiyon yazılabilir [7]. Birleştirici fonksiyon girdi olarak map fonksiyonunun çıktılarını kullanacaktır. Çıktı olarak ise reduce fonksiyonunun girdi tipinde çıktılar üretmelidir. Birleştirici fonksiyona örnek olarak aşağıdaki durum gösterilebilir:

Amacı girdi olarak verilen yıl/sıcaklık kayıtları içersinde her yılın en yüksek sıcaklıklarını belirlemek olan bir *MapReduce* işinde iki adet map görevinin 1950 yılı(anahtarı) için çıktıları(anahtar/değer) aşağıdaki gibidir. Burada map fonksiyonunun görevi sıcaklık değerlerini yıla göre gruplamaktır.

mapper-1	mapper-2
(1950, 0)	(1950, 25)
(1950, 20)	(1950, 15)
(1950, 10)	-

Aynı anahtara sahip olan çıktılar bir reduce fonksiyonuna girdi olarak verilecektir. Dolayısı ile tüm çıktı kayıtları reduce görevinin çalıştırıldığı düğüme ağ üzerinden aktarılacaktır. Reduce foksiyonunun girdi ve çıktıları aşağıdaki gibi olacaktır.

(1950, 0, 20, 10, 25, 15) → reduce fonksiyonu → (1950, 25)

Tüm map görevlerinin çıktılarını ağ üzerinde taşımak yerine map görevinin çalıştığı düğümde bir birleştirici fonksiyon çalıştırılır. Bu fonksiyon map görevinin çıktılarını alır ve o düğümdeki map çıktılarından her anahtar için en büyük sıcaklık değerini bulur. Buna göre mapper-1 için birleştirici çalıştırıldığında çıktı olarak (1950, 20) anahtar/değer ikilisi üretilecektir. Aynı birleştirici fonksiyon mapper-2 için çalıştırıldığında (1950, 25) anahtar/değer sonucunu üretecektir. Dolayısı ile reduce görevine ağ üzerinden sadece iki sonuç iletilecektir.

Birleştirici fonksiyonun çalıştırılması sonucu etkilemeyecektir. Dolayısı ile problem uzayına bağlı olarak birleştirici fonksiyon bir veya daha fazla kez çalıştırılarak ağ trafik azaltılabilir. Bu da dağıtık sistemlerde en büyük dar boğaz olan bant genişliği problemini en asgari düzeye indirecektir.

6. GEZGİN SATICI PROBLEMİNİN MAPREDUCE PGA İLE ÇÖZÜMÜ

6.1 Ortamın Hazırlanması

Geliştirilen algoritmanın çalışıtırılabilmesi için öncelikle ağ üzerindeki bilgisayarlar için gerekli *Hadoop* bileşenlerinin kurulmuş olması gerekmektedir. Sistem kurulumundan önce gerekli olan diğer yazılımlar açıklanacaktır.

6.1.1 Kurulum için gerekli programlar

Hadoop MapReduce işlerinin çalıştırılabilmesi için Cygwin, Java SDK, Ssh hizmeti ve *Hadoop* dağıtımının sisteme kurulu olması gerekmektedir.

6.1.1.1 Cygwin

Hadoop sisteminin hedef platformu linux işletim sistemidir. Dolayısı ile windows üzerinde *Hadoop* sistemini çalıştırabilmek için linux platfomunun hizmetlerini ve komutlarını sunan cygwin programını kullanmak gerekmektedir. Cygwin, internet üzerinden veya yerel olmak üzere iki farklı kurulum seçeneği sunmaktadır. Tez için yerel bir kopya oluşturulmuş ve tüm makinelere bu yerel kopya kurulmuştur. Cygwin kurulumu için 1.7.9-1 versiyonu kullanılmıştır.

Hadoop hizmetlerin bilgisayar kümesi üzerinde çalıştırılabilmesi için ssh bağlantısı kurulmalıdır. Ayrıca ssh üzerinden bağlantı kurulurken şifreli bağlantı yerine özel anahtar yöntemi ile bağlantı kurulması önerilir [9].

6.1.1.2 Java

Hadoop sisteminin tüm bileşenleri farklı bir jvm(java sanal makinesi) üzerinde çalışmaktadır. Dolayısı ile *Hadoop* sisteminin çalıştırılacağı tüm makinelerde java kurulumu gerekmektedir. Java versiyonunun Sun firmasına ait olması tercih sebebi olmalıdır. Java versiyonu olarak 1.7.02 versiyonu kullanılmaktadır.

Ayrıca Java kurulumundan sonra *Hadoop* sistemine Java konumunun belirtilmesi gerekmektedir. Bunun için JAVA_HOME ortam değişkeni tanımlanabileceği gibi, *Hadoop* konfigürasyon dosyasından da JAVA_HOME tanımlaması yapılabilir.

6.1.1.3 Eclipse

Tez kapsamında geliştirilen algoritma java programlama dili ile geliştirilmiştir. Java için en iyi geliştirme ortamlarından biri olan Eclipse, geliştirme ortamı olarak kullanılmıştır.

MapReduce programı geliştirmek için herhangi bir programlama dili ve ortamı kullanılabilir. *Hadoop* tüm diller için gerekli entegrasyonu sağlamıştır. Ancak *Hadoop* yapısı java programlama dili ile yazıldığı için Java asıl gerçekleme dili olarak sunulmaktadır. Java programı geliştirmek için ise Eclipse IDE'si kullanılmıştır. Eclipse için kullanılan versiyon Eclipse 3.3 Europa versiyonudur.

6.1.1.4 Hadoop

Hadoop kurulumu için hadoop.org sitesinden indirilen *Hadoop* versiyonu zip dosyasından herhangi bir dizine çıkartılır. *Hadoop* için 0.21.0 veriyonu kullanılmaktadır. *Hadoop* üç faklı konfigürasyonda çalışmaktadır.

Tekil mod

Bu modda tüm işlemler tek makinede ve aynı jvm üzerinde çalışır. Geliştirme için ideal bir moddur [7, 9, 10].

Psödo-dağıtık mod

Bu modda tüm hizmetler aynı bilgisayarda farklı java sanal makineleri üzerinde yerel olarak çalıştırılır. Tam dağıtık sistemi birebir simüle etmektedir. Hatta *Hadoop* tam dağıtık mod ile psödo dağıtık mod arasındaki farkı ayırt etmemektedir.

Tam dağıtık mod

Psödo-dağıtık mod ile hatalarından ayıklanan bir program daha sonra *Hadoop* sisteminin sunduğu hizmetlerden yararlanabilmesi ve gerekli performansı sağlayabilmesi için tam dağıtık modda çalıştırılmalıdır. *Hadoop* sisteminin tam dağıtık modda çalışabilmesi için ağ üzerinde en az iki makine bulunması gerekmektedir. Ağ üzerindeki makinelerden bir tanesi master makine olarak kullanılacak ve üzerinde NameNode, Secondary NameNode ve JobTraker bileşenleri çalıştırılacaktır. Kalan diğer makineler ise slave olarak kullanılacaktır. Bu makinelerde veriyi depolayan ve *MapReduce* görevlerini işleyen sırasıyla DataNode ve TaskTracker bileşenlerini çalıştıracaktır.

6.2 Kurulum ve Konfigürasyon

Hadoop sisteminin doğru şekilde çalıştırılabilmesi için öncelikle gerekli programların kurulması gerekmektedir. Daha sonra *Hadoop*'un ilgili versiyonu zip dosyasından herhangi bir dizine çıkartılır ve gerekli konfigürasyon ayarları yapılır.

6.2.1 Kurulum adımları

1- Tüm makinelere yerel bir kopyası alınan cygwin programı kurulur. Kurulum esnasında yerelde var olan tüm linux araçları ve komutları kuruluma eklenir. Bunlar arasında ağ üzerinde haberleşmeyi sağlayan ssh hizmeti de bulunmaktadır.

2- Tüm makinelerde kurulum sonrası cygwin üzerinden *Hadoop* komutlarının sorunsuz şekilde çalıştırılabilmesi için cygwin/bin ve cygwin/usr/sbin dizinleri path ortam değişkenine eklenmelidir.

3- Tüm makinelere bir önceki adımda ssh hizmeti kurulmuştu. Bu adımda ise, *Hadoop* sistemindeki master makinenin diğer makineler ile sorunsuz haberleşebilmesi için Şekil 6.1'de gösterilen şifresiz giriş ayarlarının yapılması gerekmektedir. Bunun için öncelikle bir cygwin terminali açılır ve *"$ ssh-host-config"* komutu yazılır. Komut işletilirken "privilege seperation" isteği için "no", "sshd" kurulum isteği için "yes" ve "cygwin enviroment variable" için "ntsec" girişlerinin yapılması gerekmektedir.

```
$ ssh-host-config
Generating /etc/ssh_host_key
Generating /etc/ssh_host_rsa_key
Generating /etc/ssh_host_dsa_key
Generating /etc/ssh_config file
Privilege separation is set to yes by default since OpenSSH 3.3.
However, this requires a non-privileged account called 'sshd'.
For more info on privilege separation read /usr/share/doc/openssh/RE
.

Should privilege separation be used? (yes/no) no
Generating /etc/sshd_config file
Added ssh to C:\WINDOWS\system32\drivers\etc\services


Warning: The following functions require administrator privileges!

Do you want to install sshd as service?
(Say "no" if it's already installed as service) (yes/no) yes

Which value should the environment variable CYGWIN have when
sshd starts? It's recommended to set at least "ntsec" to be
able to change user context without password.
Default is "ntsec".  CYGWIN=ntsec

The service has been installed under LocalSystem account.
To start the service, call `net start sshd' or `cygrunsrv -S sshd'.

Host configuration finished. Have fun!
```

Şekil 6.1 : Ssh sunucu konfigürasyonu.

4- Tüm makinelerde önceki adımda "sshd" servisinin kurulumunu yaptıktan sonra bu servisin başlatılması gerekmektedir. Bunun için "Bilgisayarım" ikonu üzerinde sağ tıklanarak "Yönet" seçeneği seçilir ve "Hizmetler" içersinde "CYGWIN sshd" hizmetine sağ tıklanarak "Başlat" seçeneği seçilir. Hizmet bir kere başatıldıktan sonra kendisi otomatik başlayacak ve herhangi bir işlem yapmak gerekmeyecektir.

5- Tüm makinelerde, sifresiz girişi sağlamak için açılan cygwin terminali üzerinden; *"$ ssh-keygen"* komutu yürütülür ve gelen tüm istekler "ENTER" ile geçilir. Bu adım ile her makine kendisine ait id_rsa ve id_rsa.pub adında özel ve ortak anahtar dosyalarını oluşturur.

6- Oluşturulan ortak anahtar dosyalarının isimleri makine ismi olarak düzeltilir. Örneğin master.pub, slave1.pub gibi. Bu dosyalar tüm bilgisayarlara dağıtılır. Ve tüm makinelerde her dosya için, "*$ cat xxx.pub >> authorized_keys"* komutu yürütülerek public anahtar dosyaları diğer makinelere dağıtılmış olur. Böylece ağ üzerindeki herhangi bir makine diğer makineye şifresiz olarak erişim sağlayabilmektedir.

7- Tüm makinelerde java kurulumu gerçekleştirilir. Java kurulum dizini olarak C:/Java_Hadoop ayarlanır.

8- İndirilen *Hadoop* zip dosyası cygwin/home/<kullanıcı-adı> dizini altına çıkartılır.

Adımlar sonrası gerekli tüm programlar kurulmuş durumdadır ve *Hadoop* sisteminin çalıştırılması için artık konfigürasyon ayarlarının yapılması gerekmektedir.

6.2.2 Konfigürasyon ayarları

Daha önceki kurulum adımlarında *Hadoop* ve diğer programlar için gerekli dizin yapıları belirtilmişti. Konfigürasyon ayarları sırasında kurulum dizinleri göreceli olarak belirtilecektir. *Hadoop* sisteminin tam dağıtık modda çalıştırılabilmesi için gereken konfigürasyon adımları aşağıda belirtilmiştir.

1- Öncelikli olarak *Hadoop* sisteminde çalışacak olan makinelerde c/windows/system32/drivers/ dizini altındaki host dosyası düzenlenir. Bu dosyaya master ve slave olarak çalışacak olan makineler ve bunların ip adresleri eklenir. Örn: 20.0.1.225 master, 20.0.0.223 slave1...

2- Hadoop/conf dizininde hadoop-env.sh dosyasında JAVA_HOME ortam değişkeni tanımlanır. Ve *Hadoop* sistemine bir isim verilir. (export JAVA_HOME=/cygdrive/c/Java_Hodoop ve export HADOOP_IDENT_STRING = HADOOPEE satırları eklenir.)

3- Hadoop/conf dizininde core-site.xml dosyasının içeriği tam dağıtık modda çalıştırılacak şekilde düzenlenir. <fs.default.name> parametresi ile NameNode bileşeninin hangi makinede olduğunu ve hangi porttan çalışacağını belirtir (Şekil 6.2).

```
<configuration>
     <property>
          <name>fs.default.name</name>
          <value>hdfs://master:9000</value>
     </property>
</configuration>
```

Şekil 6.2 : NameNode bilşeninin adresi.

4- Hadoop/conf dizininde core-site.xml dosyasının içeriği tam dağıtık modda çalıştırılacak şekilde düzenlenir. <dfs.replication> değeri HDFS dosya sisteminde var olan dosyaların kaç adet kopyasının oluşturulacağını belirtir (Şekil 6.3). <dfs.permissions> değeri ise windows sistemi için "erişim izni yok" hatasının oluşumunu engellemek için kullanılır.

```
<configuration>
     <property>
          <name>dfs.replication</name>
          <value>2</value>
     </property>
     <property>
          <name>dfs.permissions</name>
          <value>false</value>
     </property>
</configuration>
```

Şekil 6.3 : NameNode konfigürasyonu.

5- Hadoop/conf dizininde mapred-site.xml dosyasının içeriği tam dağıtık modda çalıştırılacak şekilde düzenlenir. <mapred.job.tracker> parametresi ile JobTracker bileşeninin çalıştırılacağı makine ve port belirtilir (Şekil 6.4).

```
<configuration>
     <property>
          <name>mapred.job.tracker</name>
          <value>master:9001</value>
     </property>
</configuration>
```

Şekil 6.4 : *MapReduce* konfigürasyonu.

6- Hadoop/bin dizininde hadoop-config.sh dosyasına CLASSPATH=`cygpath -wp "$CLASSPATH"` satırı eklenir. Bu şekilde windows işletim sistemi path ortam değişkenindeki dos formatı dizin yapısı linux işletim sistemi tarafından anlaşılır hale getirilmiş olur.

7- Master olarak belirtilen makinede hadoop/conf dizini altındaki "slaves" dosyası içersine slave olarak çalışacak makinelerin isimleri eklenir (Eklencek olan makinelerin ip-isim eşleşmeleri daha önceki adımlarda gerçekleştirilmişti).

6.3 Hadoop Sisteminin Çalıştırılması

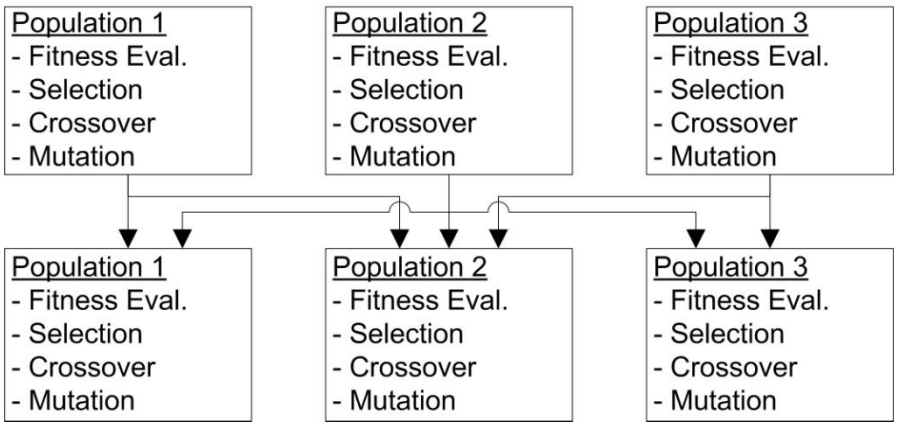
Hadoop sistemi için gerekli kurulumlar ve ayarlar yapıldıktan sonra artık sistemin çalıştırılması aşamasına geçilebilir. Bu aşamada tüm işlemler master makinede cygwin terminali üzerinden yapılır. Diğer tüm makinelerde aynı kullanıcı adı ile giriş yapılmış olmalıdır.

Öncelikli olarak "*$bin/hadoop namenode –format*" komutu ile HDFS dosya sistemi formatlanarak tüm makineler senkronize hale getirilir. Daha sonra "*$bin/start-dfs.sh*" komutu ile HDFS dosya sistemi ayağa kaldırılır. Bu aşamada NameNode, SecondaryNameNode ve DataNode bileşenleri başlatılmış olur. NameNode ve SecondaryNameNode master olarak belirtilen makinede çalıştırılır. DataNode bileşenleri ise "slaves" dosyasında belirtilen makinelerde çalıştırılır. *Hadoop* işlerinin işletilebilmesi için *MapReduce* alt sisteminin de başlatılması gerekmektedir. "*$bin/start-mapred.sh*" komutu ile JobTracker ve TaskTracker bileşenleri başlatılır ve artık *Hadoop* sistemi tam dağıtık modda çalışmaktadır.

Bundan sonraki aşamada geliştirilen *Hadoop* programı, işletilmek üzere, "$bin/hadoop jar <program-adı.jar>" komutu ile çalışmakta olan *Hadoop* sistemine iletilir.

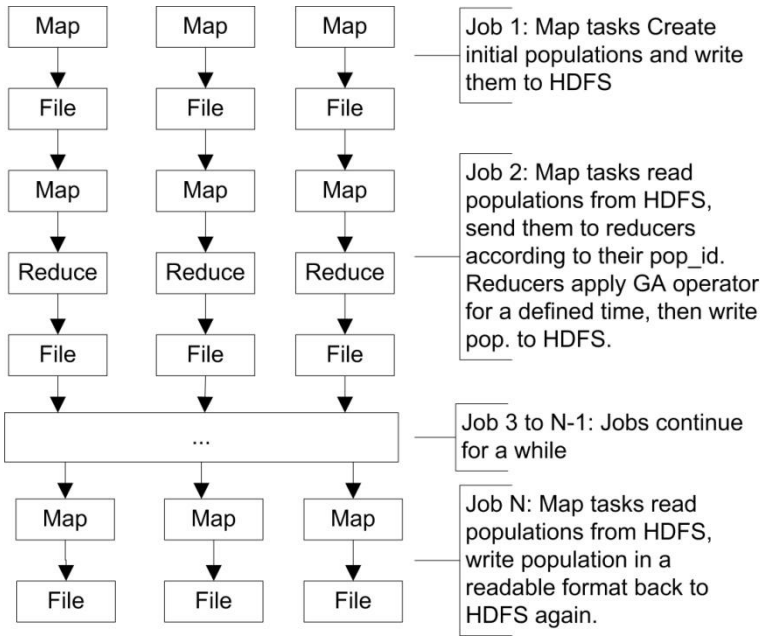
6.4 GSP Probleminin Çözümü İçin Algoritmanın Geliştirilmesi

Geliştirilen genetik algoritmanın paralelleştirilmesi için Göç alan statik alt-popülasyonlar yöntemi kullanılmıştır. Her popülasyon diğer popülasyonlardan ayrı olarak kendi içersinde gelişimini sürdürür. Şekil 6.5'de görüldüğü gibi, belli aralıklarla alt-popülasyonlar arasında bireylerin göç ettirilmesi sağlanarak alt-popülasyonlar arasında bilgi paylaşımı gerçekleştirilmiş olur.



Şekil 6.5 : Algoritmanın işleyişi.

Paralel GA'nın gerçekleştirimi için yinelemeli *MapReduce* yapısı kullanılmıştır. Her neslin gelişimi ayrı bir *Hadoop* işi olarak tanımlanmıştır. Her iş *N* (alt-popülasyon sayısı) adet map ve reduce görevlerinden oluşturulmuştur. Her iş sonunda bireyler Şekil 6.6'da açıklandığı gibi HDFS dosya sistemine yazılmaktadır.



Şekil 6.6 : *Hadoop* sisteminde gerçeklenen PGA.

Her nesil sonunda algoritmanın sonlanma koşullarını sağlayıp sağlamadığı kontrol edilir. Eğer sonlanma koşulları gerçekleşmiş ise algoritma sonlanır ve en iyi birey sonuç olarak kullanıcıya sunulur. Aksi takdirde, yeni nesil için yeni bir *MapReduce* işi başlatılır ve bir sonraki birey değişimi aşamasına kadar alt-popülasyonlar kendi içlerinde gelişmeye devam ederler.

MapReduce mimarisi, her iş sonunda sonuçların dosyaya yazılması aşamasında anahtar/değer ikilisi için varsayılan olarak tamsayı/karakter dizisi ikilisini kullanır. Ancak bu geliştirilen PGA için uygun değildir. Bunun için kromozom tipinde giriş ve çıkış formatı geliştirilmiştir. Map görevleri veriyi HDFS dosya sisteminde okurken direk olarak kromozom formatında okur ve yine reduce görevleri sonuçları HDFS dosya sistemine yazarken kromozom tipinde yazar. Bu aşamada bireylerin herhangi bir dönüşüme tabi tutulması gerekmemektedir. Bunun için geliştirilen kromozom sınıfının *Hadoop* kütüphanesinin *Writable* arayüzünü kalıtması ve *readFields()* ve *write()* fonksiyonlarını gerçeklemesi gerekmektedir. Gerçeklenen metotlar içersinde o sınıfın üye değişkenleri dosya sistemine yazılacak ve dosya sisteminde okunacak formatta belirtilmesi gerekmektedir.

MapReduce programının geliştirilmesi aşamasında Mapper ve Reducer fonksiyonlarının yanında, yazılan programın *MapReduce* sisteminde çalıştırılabilmesi için gerekli *Driver* sınıfı ve bireylerin map safhası sonunda ilgili popülasyona doğru şekilde iletilebilmesi için *Partitioner* sınıfı gerçeklenmiştir.

6.4.1 Driver sınıfı

Driver sınıfı *MapReduce* programları için ana giriş noktasıdır. *MapReduce* işlerin *Hadoop* sistemine gönderilmesi ve programın sonlandırılması işlemleri bu sınıf yardımı ile gerçekleştirilmektedir (Şekil 6.7). Ayrıca bir iş için gerekli olan map ve reduce görevlerinin sayısı, her iş için girdi/çıktı formatlarının ve dizinlerinin belirlenmesi bu sınıf yardımı ile yapılmaktadır.

```
BEGIN
   Run Job1: Creates initial population in parallel
   FOREACH max generation number
      Run Job i: evolve population
   END FOR
   Run job N: write resulting populations HDFS in a readable format to HDFS
END
```

Şekil 6.7 : Driver sınıfı.

6.4.2 Mapper fonksiyonu

Geliştirilen *MapReduce* programında iki tür mapper fonksiyonu kullanılmaktadır. Bunlarda ilki başlangıç popülasyonunun oluşturulması için kullanılmaktadır. Belirtilen sayıda mapper fonksiyonu paralel olarak her alt-popülasyon için başlangıç popülasyonunu üretir. Diğer mapper fonksiyonu ise, alt-popülasyonların evrilmesi sürecinde bireylerin HDFS dosya sisteminde okunması, popülasyon kimliklerine göre gruplanması ve reducer fonksiyonuna gönderilmesi işlemlerini gerçekleştirir.

6.4.3 Partitioner sınıfı

Hadoop sisteminin varsayılan partitioner sınıfı hash fonksiyonu kullanmaktadır [10]. Bu fonksiyon ise, bireylerin gen sıralamasını göz önüne alarak onları ilgili alt-popülasyona göndermektedir. Buna göre, benzer gen sıralamasına sahip bireyler aynı popülasyona gönderilmektedir. Bu durum, geliştirmiş olduğumuz göç alan statik popülasyonlar PGA için uygun değildir. Bunun için partitioner sınıfı yeniden düzenlenmiştir. Gerçeklenen yeni partitioner sınıfında bireyler taşımış oldukları popülasyon kimliklerine göre ilgili alt-popülasyona gönderilmektedir. Sonuç olarak aynı popülasyon kimliğini taşıyan tüm bireylerin ilgili alt-popülasyonda yer alması sağlanmıştır.

6.4.4 Reducer fonksiyonu

Reducer fonksiyonu (Şekil 6.8), aynı popülasyon kimliğine sahip bireyleri girdi olarak alır. Tüm bireylerin alt-popülasyona eklenmesinden sonra, popülasyon belirlenen iteresyan sayısı kadar evrilir. İlk olarak bireylerin uygunluk değerleri hesaplanır. Rank seçimi ile çaprazlama için uygun olan bireyler seçilir. Aç-gözlü çaprazlama yöntemi kullanılarak, yeni bireyler elde edilir. Elde edilen yeni bireyler üzerinde belirli olasılık dahilinde mutasyon işlemi uygulanır. Böylece yeni bir jenerasyon elde edilmiş olur. Belirtilen iterasyon sayısı sonunda bireyler tekrar HDFS dosya sistemine yazılır. Buradaki önemli nokta bireyleri anahtar/değer (popülasyon kimliği/kromozom) ikilisi olarak yazarken, en iyi bireylerin popülasyon kimliği olarak tüm alt-popülasyonlarının kimliklerinin yazılmasıdır. Böylece bir sonraki *MapReduce* işinde, map görevi bireyleri gruplarken, en iyi bireyler tüm alt-popülasyonların kimliğine sahip olduklarından, onları tüm reducer fonksiyonlarına(alt-popülasyonlara) gönderecektir. Bu sayede popülasyonlar arasında en iyi bireylerin göç ettirilmesi sağlanmış olur.

```
BEGIN
   Receive individuals that all have the same population id
   FOREACH evolution number
      FOREACH population size
         Apply rank selection
         Apply Greedy crossover
         Add new individuals to new population
      END FOR
      Apply mutation to new population
   END FOR
   Write population to HDFS
   FOREACH number of other sub-populations
      Change individual pop_id and write it to HDFS
   END FOR
END
```

Şekil 6.8 : Reducer sınıfı.

Her alt-popülasyon için ayrı bir map/reduce görevi atanmıştır. Her görev dağıtık sistem üzerinde paralel olarak işlemekte ve alt-popülasyonlar birbirinden bağımsız olarak gelişmektedir. Belirtilen iterasyon sonunda bireyler HDFS dosya sistemine yazılırken, en iyi bireyler ayrıca başka bir dosyaya daha kaydedilmektedir. Driver sınıfı yeni bir *MapReduce* işi başlatmadan önce en iyi bireylerin yazılı olduğu dosyayı kontrol eder. Eğer dosyada belirtilen kriteri sağlayan bir birey varsa program çalıştırmayı durdurur ve o bireyi problemin çözümü olarak kullanıcıya sunar.

6.5 Geliştirilen Algoritmanın Analizi

Geliştirilen paralel genetik algoritmanın performansını ölçmek için ardışıl GA ile karşılaştırılmıştır. Geliştirilen ardışıl genetik algorimanın sonuçları ise daha önce gerçeklenen diğer GA'lar ile karşılaştırılmıştır.

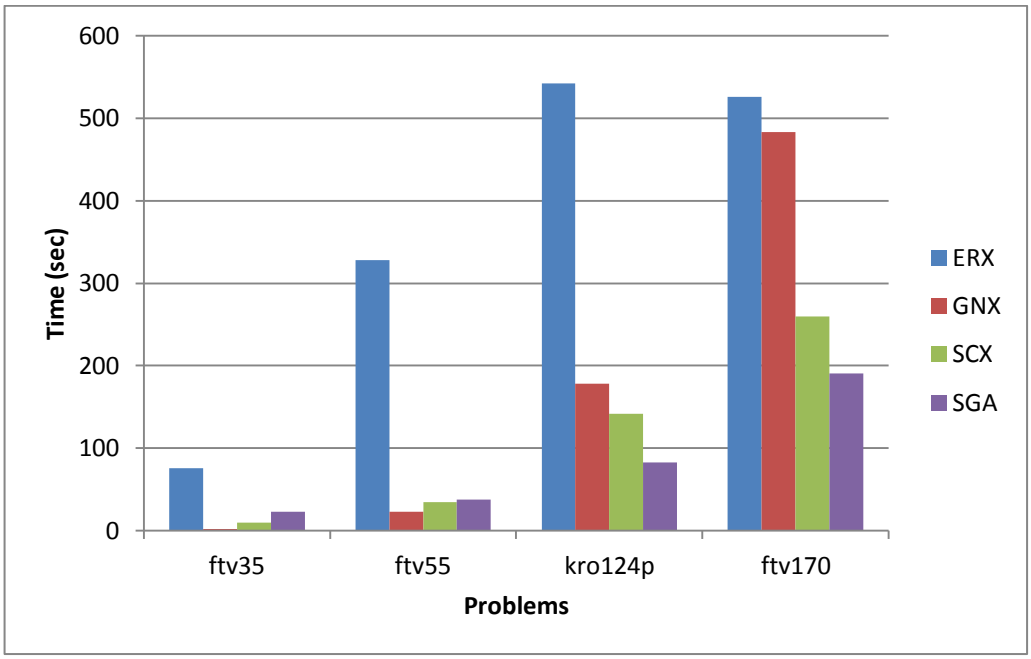
Ardışıl GA, Intel Core Duo 2.4 GHz işlemic ve 3GB RAM'a sahip bir makine üzerinde çalıştırılmıştır. Geliştirilen PGA ise 6 makineden oluşan ve özellikleri Çizelge 6.1'de verilen bir *Hadoop* ağı üzerinde çalıştırılmıştır.

Çizelge 6.1 : Hadoop ağı.

Bilgisayar adı	İşlemci Gücü(GHz)	Çekirdek sayısı	İşlemci Tipi	Bellek (GB)
Master	3.0	2	Pentium-4	2.0
Slave1	3.2	2	Pentium-4	2.0
Slave2	3.0	2	Pentium-D	2.0
Slave3	2.13	2	Core-duo	3.0
Slave4	2.33	4	QuadCore	3.5
Slave5	2.8	4	İ5	2.5

Kullanılan tüm problemler TSPLIB [29] kütüphanesinin sunmuş olduğu problemlerdir. Tüm problemler asimetriktir, buna göre iki şehir arasındaki gidiş mesafesi ve geliş mesafesi farklı olabilmektedir. Sonuçların elde edilebilmesi için tüm koşumlar en az 10 kez tekrarlanmıştır. Popülasyon büyüklüğü ardışıl algoritma için 100, *Hadoop* üzerinde çalışan algoritma için her alt-popülasyon için 100 olarak belirlenmiştir. Maksimum iterasyon sayısı her koşum için 50.000 olarak ayarlanmıştır. Çaprazlama oranı 0.99, mutasyon oranı 0.3 olasılıklıdır. Yüzde hesabı için kullanılan denklem şu şekildedir. $Yüzde = \frac{Bulunan\ sonuç - Optimum\ Sonuç}{Optimum\ Sonuç} \times 100.$ Bu şekilde bulunan sonucun optimum sonuctan ne kadar saptığı yüzde olarak ifade edilmektedir.

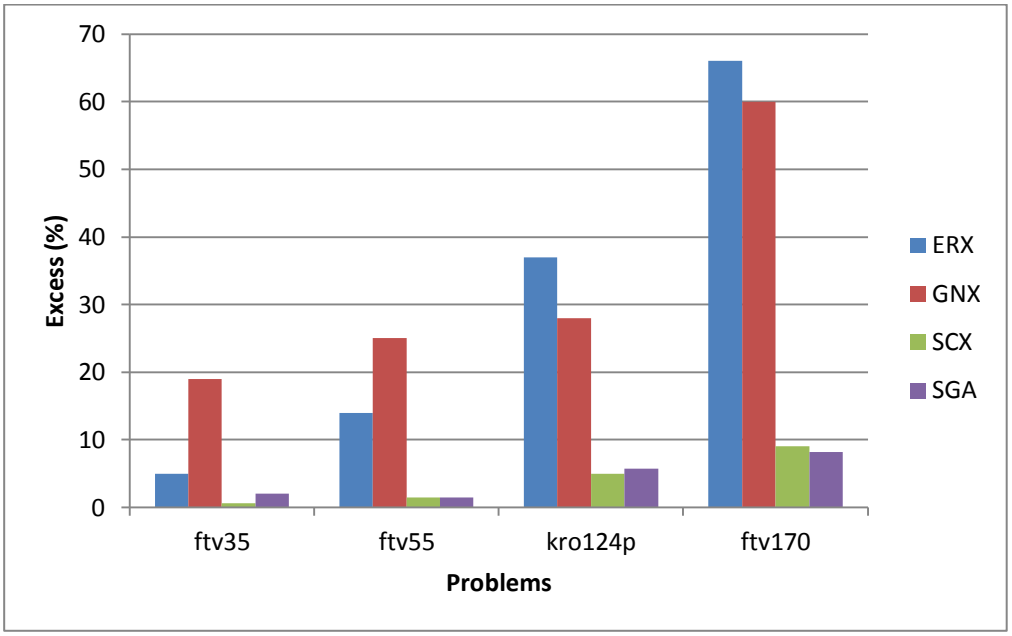
İlk olarak geliştirilen ardışıl GA, daha önceki çalışmalar ile karşılaştırılmıştır. Bu kapsamda, kenar sıralama çaprazlaması (Edge Recombination Crossover - ERX), genelleştirilmiş *N*-noktalı çaprazlama (Generalized N-point Crossover - GNX) ve sıralı yapısal çaprazlama (Sequential Constructive Crossover - SCX) yöntemlerini kullanan genetik algoritmalar [30], geliştirilen SGA ile karşılaştırılmıştır. Algoritmaların ilgili problemler için ne kadar zamana ihtiyaç duydukları ve her bir algoritmanın optimum çözümden ne kadar saptığı Şekil 6.9 ve Şekil 6.10'da verilmiştir. Bu karşılaştırmada nesil sayısı, daha önce yapılan çalışmalarda kullanılmış nesil sayısı olan, 10.000 olarak kullanılmıştır.



Şekil 6.9 : Ardışıl GA zaman karşılaştırmaları.

Daha önce yapılan çalışmalarda çözüme ulaşmak için gerekli iterasyon sayısı hakkında bir bilgi olmadığı için, en iyi bireyin kaçıncı iterasyon sonrasında bulunduğuna dair bir karşılaştırma gerçekleştirilememiştir. Bunun yanında, her bir problem için en iyi çözümün belli olması nedeni ile zaman açısından karşılaştırma yapılırken, algoritmaların belirlenen en iyi sonucu bulmaları ya da maksimum iterasyon sayısına ulaşıncaya kadar çalışmaları beklenmiştir.

Algoritmaların belirtilen problemler için harcadıkları zamanlar Şekil 6.9'da belirtilmişir. Buna göre ERX algoritması en kötü sonucu verirken, tez kapsamında geliştirilen algoritmaya en yakın sonuçlar veren SCX algoritması olmuştur. Ancak problem büyüklüğü arttıkça geliştirilen SGA tüm algoritmalardan daha kısa sürede sonuç üretmiştir.



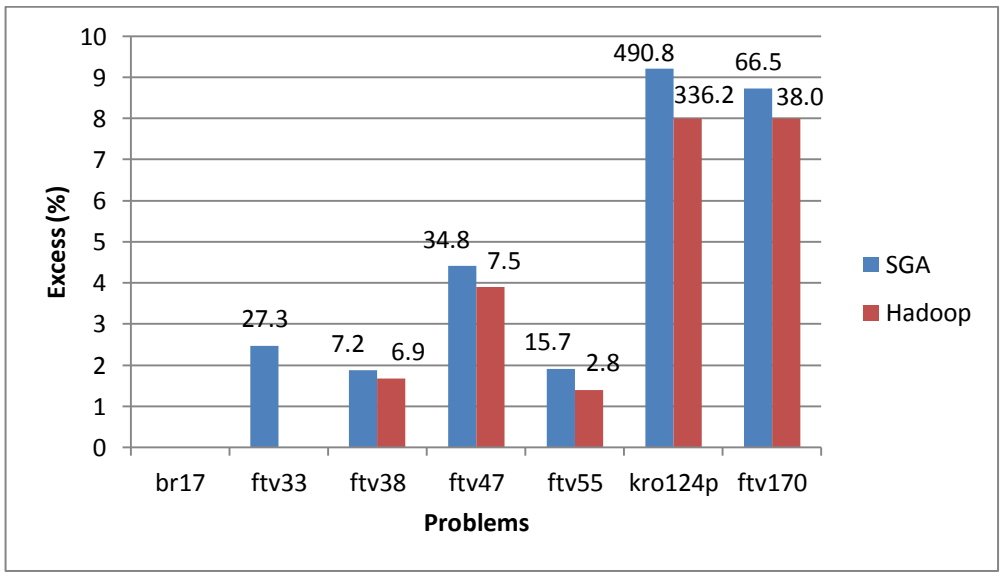
Şekil 6.10 : Ardışıl GA yüzde olarak optimum çözümden sapmalar.

Geliştirilen SGA ve diğer algoritmaların her problem için optimum çözümden ne kadar saptıkları yüzde olarak Şekil 6.10'da gösterilmiştir. ERX ve GNX algoritmaları problem boyutu arttıkça optimum sonuçtan uzaklaşmaktadır. SGA ve SCX algoritmaları ise birbirne çok yakın sonuçlar üretmektedir. Sonuçlara bakıldığında geliştirilen SGA, diğerlerine göre daha iyi çözümü yine diğer algoritmalardan daha kısa sürede üretmektedir.

Bir sonraki karşılaştırma SGA ile *Hadoop* sistemi üzerinde çalıştırılan PGA arasında gerçekleştirilmiştir. Bu karşılaştırmada maksimum nesil sayısı 50.000 olarak ayarlanmıştır.

Her ne kadar Ardışıl GA ile *MapReduce* PGA toplam birey bakımından eşit sayıda bireye sahip olsalar da, *MapReduce* PGA'da her alt-popülasyon çözüm uzayında farklı bir yöne doğru arama yaptığından, Şekil 6.11'de görüldüğü gibi, her zaman SGA'ya göre daha iyi çözümler üretmektedir.

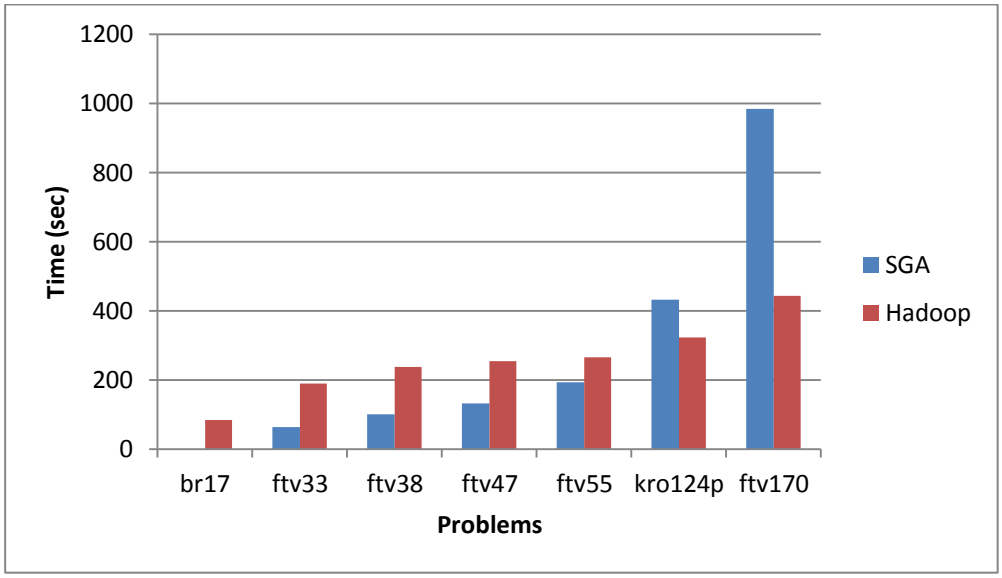
Ayrıca Şekil 6.11'de tüm barların üzerinde, her problem için Ardışıl GA ve *MapReduce* PGA'nın standart sapmaları verilmiştir. Buna göre her koşum sonrasında elde edilen sonuçların birbirinden ne kadar farklılık gösterdiği tespit edilebilmektedir. Sonuçlara bakıldığında *MapReduce* PGA'nın tüm problemler için daha düşük standart sapmaya sahip olduğu görülmektedir. Bu da, *MapReduce* PGA'nın her seferinde farklı yönlerde arama yaparak optmimuma en yakın sonucu elde ettiğini, ancak Ardışıl GA'nın her seferinde çözüm uzayında farklı yönde arama yaptığını ve farklı sonuçlar elde ettiğini göstermektedir.



Şekil 6.11 : SGA ve PGA optimum çözümden sapmalar.

Şekil 6.12'de PGA ve SGA'nın problem büyüklüğüne göre harcadıkları zaman gösterilmektedir. Problem büyüklüğünün az olduğu problemler için SGA daha kısa sürede çözüm üretmektedir. Bunun nedeni *Hadoop* mimarisinin her görev için farklı bir java sanal makinesi (Java Virtual Machine – JVM) oluşturmasından ve her *Hadoop* işi için verilerin HDFS dosya sisteminden okunması ve HDFS dosya sistemine iş tamamlandıktan sonra yazılmasından kaynaklanmaktadır. Küçük problemler için *MapReduce* PGA'da JVM oluşturulması, *Hadoop* işi için gerekli olan verilerin ağ üzerinden okuması ve iş sonlandıktan sonra sonuçların tekrar dağıtık dosya sistemine yazılması işlemlerinin gerçekleştirilmesi çözüm zamanını

domine etmektedir. Ancak problem büyüklüğü arttıkça; PGA'nın çözüm süresi çok fazla değişmezken, SGA'nın çözüm süresi dramatik bir şekilde artış göstermektedir. Analizler geriye dönük yapılmış olan çalışmaları içerdiği için ve bu çalışmalarda bir *Hadoop* işi için her aşamanın ne kadar süre aldığı tespit edilmediği için girdi/çıktı işlemleri ve sanal makine oluşturma işlemlerinin ne kadar süre aldığı kesin olarak belirtilmemiştir.

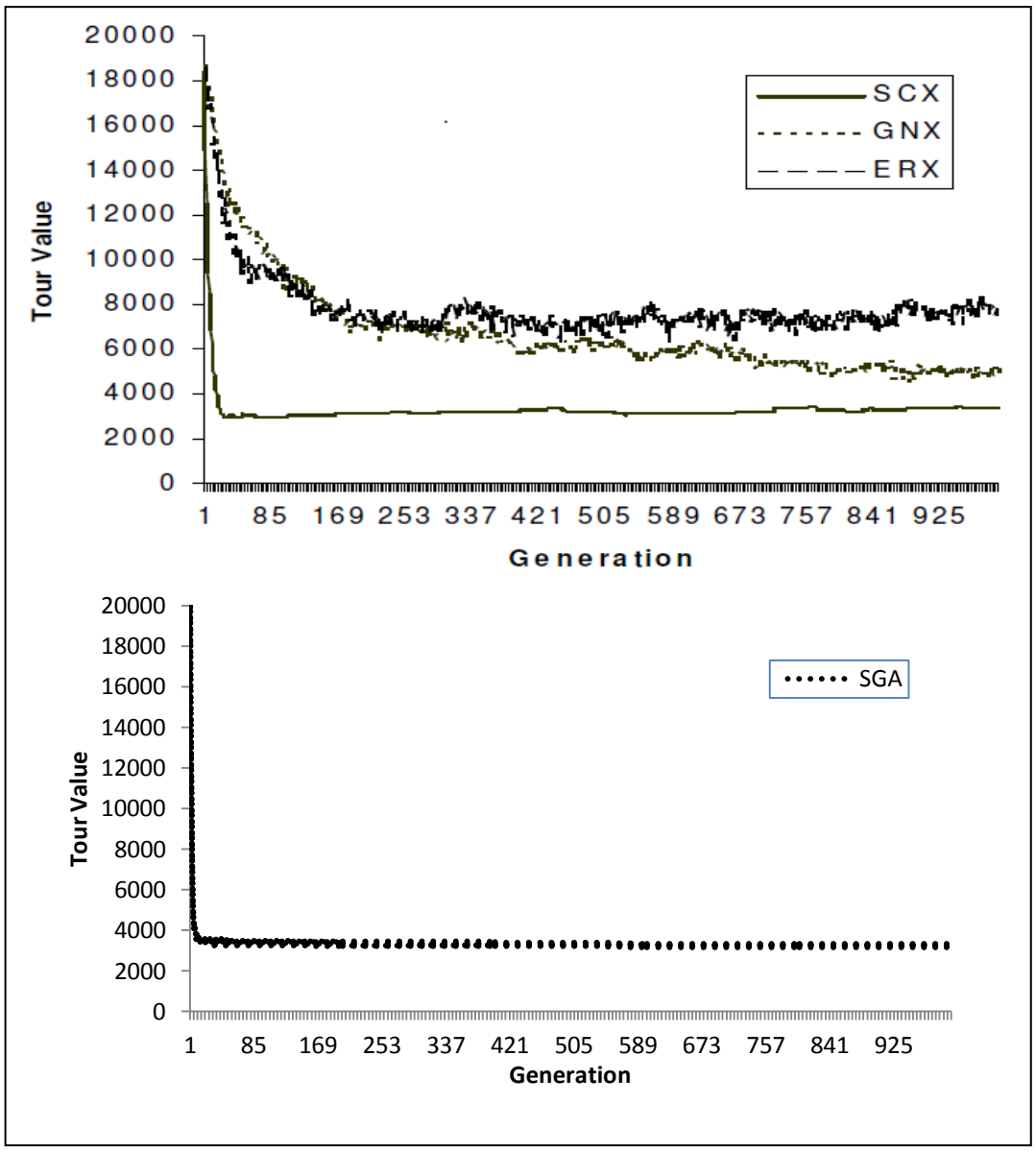


Şekil 6.12 : SGA ve PGA zaman karşılaştırması.

Ayrıca Şekil 6.13'te daha önceki çalışmalar ile geliştirilen ardışıl GA ve MapReduce PGA algoritmalarının 171 adet şehirden oluşan bir problem için gösterdikleri performanslar sergilenmektedir. Her algoritmanın 1000 nesil sonrasında yakınsama sonuçları görülmektedir. Daha önceki çalışmalarda 1000 nesile kadar olan kısım analiz edildiği için bu çalışmada da ilk 1000 nesil için analiz gerçekleştirilmiştir.

Her çaprazlama operatörü, dolayısı ile geliştirilen her algoritma belli bir rastsallık içermektedir. Bu da bu algoritmaların optimizasyon problemlerinde kullanılmalarındaki en büyük etkenlerden birisidir. Bir çaprazlama operatörü ne kadar yüksek rastsallık oranına sahip ise, o oranda çözümün iyileştirilmesi süreci devam eder. Şekil 6.13'e bakıldığında, GNX algoritmasının çözüm çeşitliliği yüksek olmasına rağmen en iyi sonucu vermemektedir. ERX algoritması bir miktar çözüm çeşitliliği sunmakta ancak en kötü sonucu vermektedir. SCX ve geliştirilen GA, en iyi yakınsama sonucuna erişmektedirler. Ancak, bu iki algoritma da çok fazla çeşitlilik sunmamakta ve yerel optimum çözüme takılma sorunuyla karşılaşmaktadırlar. Geliştirilen MapReduce PGA yardımı ile her alt-popülasyon

çözüm uzayını farklı yönlerde taramaktadır. Her iş sonrasında belirlenen sayıdaki birey alt-popülasyonlar arasında paylaşılmaktadır. Bu yöntem yerel optimuma takılma sorunu çözmektedir.



Şekil 6.13 : 170 şehirli bir problem için algoritmaların yakınsama grafikleri.

7. SONUÇ VE ÖNERİLER

Gezgin satıcı problemi, günümüzde optimizasyon problemleri için geliştirilen algoritmaların performansının test edilmesi sürecinde sıklıkla kullanılmaktadır. Ayrıca devre tasarımı ve lojistik başta olmak üzere birçok alanda uygulama bulması nedeni ile GSP önemli bir problem olarak karşımıza çıkmaktadır.

Küçük boyuttaki GSP için kesin çözüm yöntemlerinin bulunmasına karşılık problem büyüklüğünün artması ile beraber kesin sonuç üreten algoritmalar işlevsiz hale gelmekte veya çözüm için çok fazla zamana ihtiyaç duymaktadır. Bu nedenle optimum sonuca belirli bir yakınlıkta çözüm üreten ve bunu kabul edilebilir bir zaman aralığında yapabilen birçok algoritma geliştirilmiştir. Tez kapsamında bu algoritmalardan biri olan genetik algoritma kullanılmıştır.

GA, optimizasyon problemlerinin çözümünde sıklıkla kullanılan ve kısa sürede optimum sonuca yakın sonuçlar üretebilen bir algoritmadır. GA'nın anlaşılmasının ve gerçeklenmesinin kolay olması, optimizasyon alanında çalışan birçok araştırmacının dikkatini çekmiştir. En önemli uygulama alanları ise, optimizasyon, mekanik öğrenme, üretim hattı çizelgeleme ve gezgin satıcı problemidir.

GA, problemin çok büyük olduğu durumlarda yavaş kalabilmektedir. Bazı durumlarda ise çözüm uzayının yeterli derecede taranamaması nedeni ile yerel optimum sonuca takılabilmektedir. Bu gibi sorunlar için GA'nın paralelleştilmesi çözüm olmaktadır. Paralel GA çözüm süresini kısaltmasının yanında çözüm uzayını da farklı yönlerde arayabilmesi nedeni ile daha iyi sonuçlar elde etmektedir.

GA'nın paralelleştirilmesi sürecinde birçok yöntem bulunulmaktadır. En sık kullanılan yöntemlerden bazıları; uygunluk değerinin hesaplanmasının paralel olarak gerçekleştirilmesi, popülasyonun alt-popülasyonlara ayrılarak her alt-popülasyonun farklı yönlerde ve paralel olarak evrimleşmesi yöntemleridir. Uygunluk hesabının paralelleştilmesi durumunda paralel GA daha hızlı çözüm üretse de, ardışıl GA'dan daha iyi bir sonuç üretmeyi garanti etmez. Ancak alt-popülasyon yönteminde hem hız hem de çözüm kalitesi daha iyi olacaktır.

Alt-popülasyonların farklı makinelerde çalışabilmesi için dağıtık mimari kullanılmaktadır. Tez kapsamında dağıtık mimari olarak *MapReduce* mimarisi kullanılmıştır. Her ne kadar *MapReduce* mimarisi veri-yoğun işlemler için daha uygun olsa da işlemci-yoğun algoritmalar içinde iyi bir alternatif oluşturmaktadır. Bu çalışmada, *MapReduce* mimarisinin ücretsiz lisanslı gerçekleştirimi olan *Hadoop* kullanılmıştır. *Hadoop* geliştirilen algoritma için *MapReduce* mimarisi sağlarken, verilerin saklanması için de HDFS sistemini kullanıcıya sunmaktadır. HDFS kullanımı ile beraber hem verilere hızlı erişim sağlanmakta hem de verilerin ağdaki makinelerde yedekli tutulması sayesinde veri kaybının önüne geçilmiş olmaktadır.

Tez çalışması ile *MapReduce* mimarisinin GA'nın paralelleştirilmesi sürecinde nasıl kullanılabileceği araştırılmış ve geliştirilen algoritmanın GSP çözümünde nasıl kullanılabileceği ortaya konmuştur. Ayrıca *MapReduce* alt yapısının sadece veri-yoğun işlerde değil, işlemci-yoğun işlerde de iyileştirme sağladığı gösterilmiştir.

MapReduce mimarisinin yeni gelişen teknolojiler arasında yer alması ve işlemci-yoğun işlerde kullanımı konusunda çok fazla çalışma yapılmış olmaması nedeni ile geliştirme aşamasında çok fazla zorluklar ile karşılaşılmıştır. Bunun yanında *Hadoop*'un Linux ortamında geliştirilmiş olması ve Windows platformu için çok fazla destek sunmaması nedeni ile sistemin Windows işletim sistemi üzerinde yönetimi için çok fazla çaba harcanmıştır. Hazır bir *Hadoop* sisteminin kurulu olmaması nedeni ile algoritma geliştirmenin yanında sistem bakımı, onarımı gibi işlemler ile de uğraşılmak zorunda kalınmıştır.

Yapılan çalışma ile *Hadoop MapReduce* programlama modeli açıklanmış, HDFS ile *MapReduce* arasındaki ilişki ortaya konmuştur. Paralel GA'nın *MapReduce* mimarisine uygun şekilde nasıl gerçeklenebileceği gösterilmiştir. GSP çözümü için GA kullanılmış ve bu algoritmanın nasıl paralelleştirilebileceği gösterilmiştir. Daha önceki çalışmalar, GA'nın *MapReduce* mimarisinde paralelleştirme yöntemi olarak uygunluk fonksiyonunun paralelleştirilmesi yöntemini kullanmışlardır. Bu çalışmada ise alt-popülasyonlara ayırma yöntemi kullanılmıştır. Böylece sadece zaman olarak değil, çözüm kalitesinin de iyileştirilmesi sağlanmıştır.

Yapılan bu çalışmanın ülkemizin yeni teknolojilere ayak uydurabilmesi ve mühendislik alanındaki gelişmeleri takip edebilmesi için iyi bir çıkış noktası olacağı

düşünülmektedir. Bundan sonraki yapılacak çalışmalar için iyi bir kaynak olması beklenmektedir.

Bundan sonraki çalışmalarda, *Hadoop* açık kaynak kodunun değiştirilerek işlemci-yoğun işler için daha uygun bir alt yapı haline getirilmesi planlanmaktadır. Ayrıca *Hadoop* ağına daha fazla makine eklenerek, çok daha büyük problemler için vereceği sonuçlar incelenecek ve daha önceki çalışmaların sonuçları ile karşılaştırılacaktır.

KAYNAKLAR

[1] **Melanie, M.** (1999). An Introduction to Genetic Algorithms (5. baskı). *Bradford Book The MIT Press*. Cambridge, Massachusetts. London, England. 978-0-262-13316-6.

[2] **Er, H. R. ve Erdoğan, N.** (2013). Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster. *The International Conference on Soft Computing and Software Engineering*. San Francisco, CA, 1-2 Mart.

[3] **Borovska, P.** (2006). Solving the Travelling Salesman Problem in Parallel by Genetic Algorithm on Multicomputer Cluster. *The International Conference on Computer Systems and Technologies*. University of Veliko Tarnovo, Bulgaria, 15-16 Haziran.

[4] **Matai, R., Mittal, M. L., Singh, S. P .**(2010). Traveling Salesman Problem, Theory and Applications. CC BY-NC-SA. India. 978-953-307-426-9.

[5] **EMEL, G. G. ve Taşkın, Ç.** (2002). Genetik Algoritmalar ve Uygulama Alanları. *Uludağ Üniversitesi, İktisadi ve İdari Bilimler Fakültesi Dergisi*. Cilt XXI, Sayı 1, s. 129-152.

[6] **Nowostawski, M. ve Poli, R.** (1999). Parallel Genetic Algorithm Taxonomy. *Third International Conference on Knowledge-based Intelligent Information Engineering Systems*. Adelaide, Australia, 31 Ağustos – 1 Eylül.

[7] **White, T.** (2009). Hadoop: The Definitive Guide. *O'Reilly Media, Inc*. USA. 978-0-596-52197-4.

[8] **Campbell, R. H., Goldberg, D. E., Llora, X., Verma, A.** (2009). Scaling Genetic Algorithms using MapReduce. *Ninth International Conference on Intelligent Systems Design and Applications*. Pisa, Italy, 30 Kasım – 2 Aralık.

[9] **Lam, C.** (2010). Hadoop in Action. *Manning Publications Co*. USA. 9781935182191.

[10] **Venner, J.** (2009). Pro Hadoop. *Apress Media LLC*. New York, NY. 978-1-4302-1942-2.

[11] **Yang, F.** (2010). Solving Traveling Salesman Problem Using Parallel Genetic Algorithm and Simulated Annealing. *Massachusetts Institute of Technology*. Cambridge, MA, 18 Mayıs.

[12] **Chiang, M. C., Tsai, C. W., Tseng, S. P., Yang, C.S.** (2010). A Fast Parallel Genetic Algorithm for Traveling Salesman Problem. *The Second Russia-Taiwan conference on Methods and tools of parallel programming multicomputers*. Vladivostok, Russia, 16-19 Mayıs.

[13] **Sadasivam, G. S. ve Selvaraj, D.** (2010). A Novel Parallel Hybrid PSO-GA using MapReduce to Schedule Jobs in Hadoop Data Grids. *Second World Congress on Nature and Biologically Inspired Computing*. Kitakyushu, Fukuoka, Japan, 15-17 Aralık.

[14] **Kureichick, V. M., Miagkikh, V. V., Topchy, A. P.** (1996). Genetic Algorithm for Solution of the Traveling Salesman Problem with New Features against Premature Convergence. *Taganrog State University of Radio-Engineering*. Taganrog, Russia.

[15] **Sudhakar, V. J.** (2011). A New Approach to Solve the Classical Symmetric Traveling Salesman Problem by Zero Suffix Method. *Int. J. Contemp. Math. Sciences, Vol. 6, 2011, no. 23, 1111 – 1120.*

[16] **Gharan, S. O. ve Saberi, A.** (2011). The Asymmetric Traveling Salesman Problem on Graphs with Bounded Genus. *The Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. San Francisco, CA, USA, 23-25 Ocak.

[17] **Chalaturnyk, A.** (2008). A Fast Algorithm For Finding Hamilton Cycles. *Thesis presented to the University of Manitoba in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science*. Winnipeg, Manitoba, Canada.

[18] **Gutin, G. ve Yeo, A.** (2007). The Greedy Algorithm for the Symmetric TSP. *Algorithmic Operations Research, Vol. 2, No. 1, 2007, p. 33-36.*

[19] **Liang, K. Y. ve Zhang, X.** (2002). Kernel Nearest-Neighbor Algorithm. *Journal Neural Processing Letters, Volume 15 Issue 2, p. 147-156*. Hingham, MA, USA.

[20] **Matsui, T.** (1994). The Minimum Spanning Tree Problem on a Planar Graph. *Department of Mathematical Engineering and Information Physics Faculty of Engineering, University of Tokyo*. Tokyo, Japan.

[21] **Gomez, P. A., Hougen, D. F.** (2007). Initial Population for Genetic Algorithms: A Metric Approach. *International Conference on Genetic and Evolutionary Methods*. Las Vegas, Nevada, USA, 25-28 Haziran.

[22] **Potvin, J. Y.** (1996). Genetic Algorithms for the Traveling Salesman Problem. *Annals of Operations Research 63*, 339-370.

[23] **Bolat, B., Erol, K. O., İmrak, C. E.** (2004). Genetic Algorithms in Engineering Applications and The Function Of Operators. *Sigma Journal of Engineering and Natural Sciences*. 2004/4.

[24] **Goldberg, D.E.** (2006). Genetic Algorithms in Search, Optimization and Machine Learning (28. baskı). *Addison Wesley Longman Inc*. USA. 0201157675.

[25] **Anand, V., Spears, W. M.** (1991). A Study Of Crossover Operators in Genetic Programming. *The 6th International Symposium on Methodologies for Intelligent Systems*. Charlotte, N:C, USA, 16-19 Eylül.

[26] **Chen, S. ve Smith, S. F.** (1996). Commonality and Genetic Algorithms. *The Robotics Institute Carnegie Mellon University*. Pittsburgh, Pennsylvania, Aralık 1996.

[27] **Altıntaş, C.** (2011). Sezgisel Algoritmalarla Sınav Çizelgeleme Problemi Çözümü. *Süleyman Demirel Üniversitesi, Bilgisayar Mühendisliği Anabilim Dalı, Yüksek Lisans Tezi*. Isparta.

[28] **Yang, R.** (1997). Solving Large Travelling Salesman Problems with Small Populations. *Department of Computer Science, University of Bristol*. U.K.

[29] **TSPLIB** (2012). <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>, alındığı tarih: 14.08.2012

[30] **Ahmed, Z. H.** (2010). Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator. *International Journal of Biometrics and Bioinformatics, Volume 3 Issue 6, p. 96-105.*

ÖZGEÇMİŞ

Ad Soyad: Harun Raşit Er

Doğum Yeri ve Tarihi: Akhisar, 1986

E-Posta: harunrasiter@gmail.com

Lisans: Ege Üniversitesi

Mesleki Deneyim ve Ödüller:

2008 – 2012 Tübitak Bilişim Teknolojileri Enstitüsü

2012 – 2013 Tübitak İLTAREN

TEZDEN TÜRETİLEN YAYINLAR/SUNUMLAR

- **Er, H. R. ve Erdoğan, N.** (2013). Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster. *The International Conference on Soft Computing and Software Engineering*. San Francisco, CA, 1-2 Mart.