Gebze Technical University

Computer Engineering

CSE 424 - 2019 Fall

HOMEWORK 1 - PART 2 REPORT

MUHAMMED OKUMUŞ

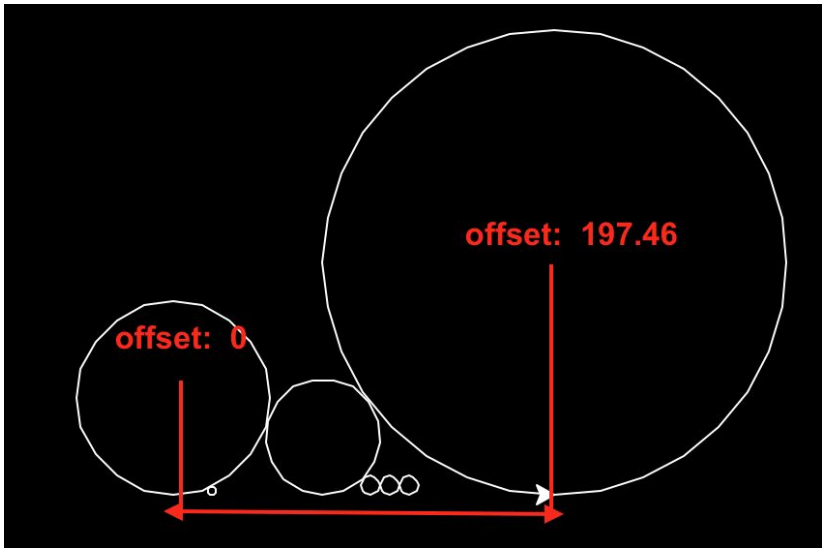
151044017

İlk Ödevden Taşınan Değişiklikler

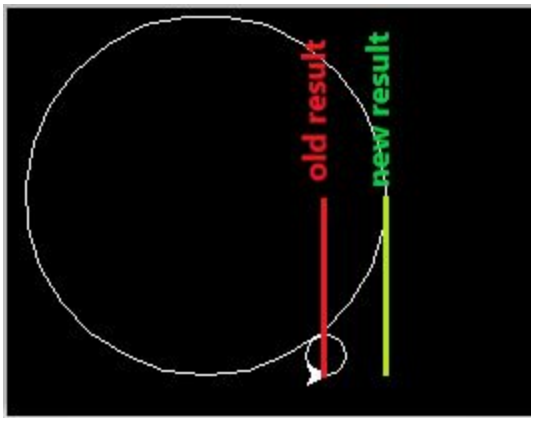
Cost Function

İlk versiyon cost fonksiyonu sadece ilk ve son çemberin yere dokunduğu noktaları hesaba katıyordu. Bu yaklaşımda kritik bir hata olduğunu fark ettim. Eğer son çember, ondan önce gelen çemberden 4 kat küçük ise, önce gelen çemberin yarıçapı, son çemberin yere dokunma noktasını geçecektir. Resimlerle ifade ederek buradaki sorunu daha iyi anlayabiliriz.

İlk versiyonumuz:



Düzeltilmiş hali:



Kısaca cost fonksiyonu üstteki görseldeki bir case'de de çalışacak şekilde tekrar yazıldı.

Tabu Search

Strateji

Tabu search mantığı, yakın zamanda ziyaret ettiğimiz çözümleri tekrar üretmemeye dayalı. Bu yapının sağlanması için üretilen çözümler belli bir kritere göre tabu olarak tutuluyor. Böylece algoritma, arama uzayında farklı yönlere gidebiliyor.

Pseudocode

Bu implementasyonda sadece short-term memory kullanan ve cost fonksiyonunu minmize eden bir tabu search gerçeklendi. İlk çözüm ise part-1 de yazılan greedy metod ile oluşturuldu.

$Input : TabuList_{size}\ ,\ MaxIterations$

$Output :\ S_{best}$

$S_{best} \leftarrow Greedy\ Solution$

$TabuList \leftarrow \oslash$

$While\ (MaxIterations\ not\ reached)$

$\quad Candidates \leftarrow \oslash$

$\quad For\ (S_{candidate} \in NonTabuNeighbours)$

$\quad\quad Candidates \leftarrow S_{candidate}$

$\quad If\ (LocateBest(\ Candidate\ )\ is\ better\ than\ \ S_{best})$

$\quad\quad Update\ \ S_{best}\ with\ Candidate_{best}$

$\quad\quad TabuList \leftarrow\ S_{best}$

$\quad\quad If\ (size(TabuList)\ >\ TabuList_{size})$

$\quad\quad\quad TabuList \leftarrow remove\ oldest\ tabu$

$return\quad S_{best}$

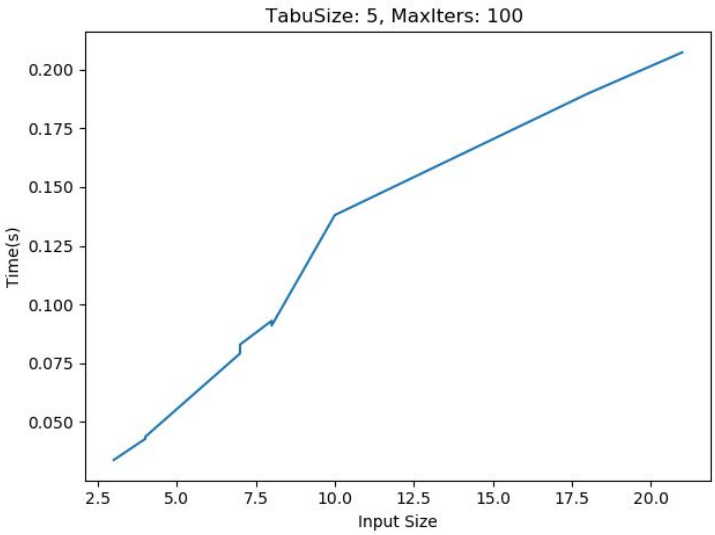
Complexity

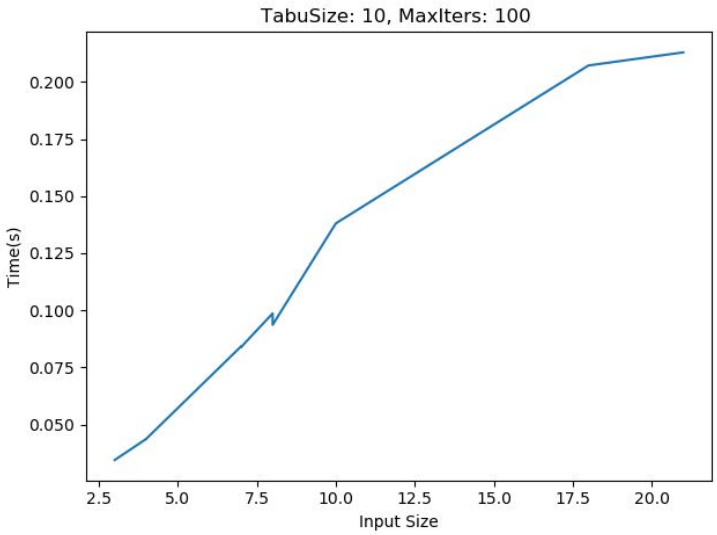
Tabu search'de fonksiyon gövdesi iç içe 2 döngüden oluşmakta. Dıştaki while döngüsü MaxIteraration inputu kadar dönmekte. Onun içerisindeki For döngüsü ise NonTabuNeighbour sayısı kadar dönmekte. NonTabuNeighbour sayısı bu implementasyonda 3 olarak belirlendi. Yani her iterasyonda maksimum üretebileceğimiz candidate sayısı 3. Bu adayların hepsi farklı veya aynı olabilir. Dış döngü n içerideki döngü ise k olmak üzere, zaman karmaşıklığımız $O(n * k)$ olur. Yani tabu search lineer sürede çalışmaktadır.

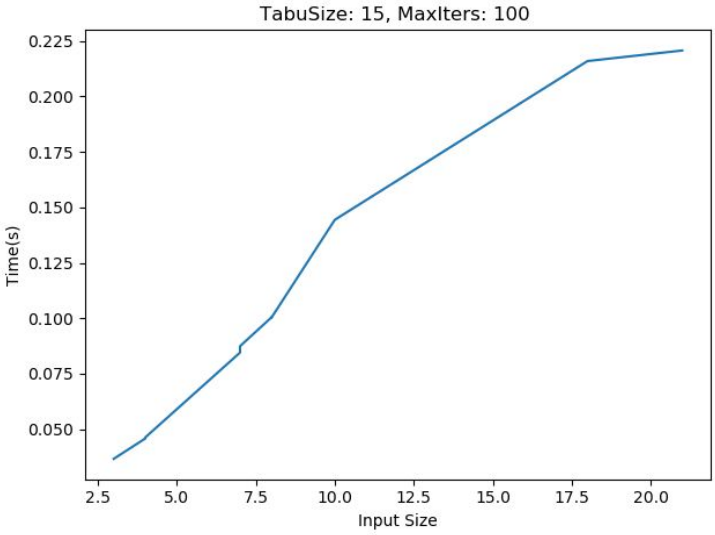
Parametre Testleri

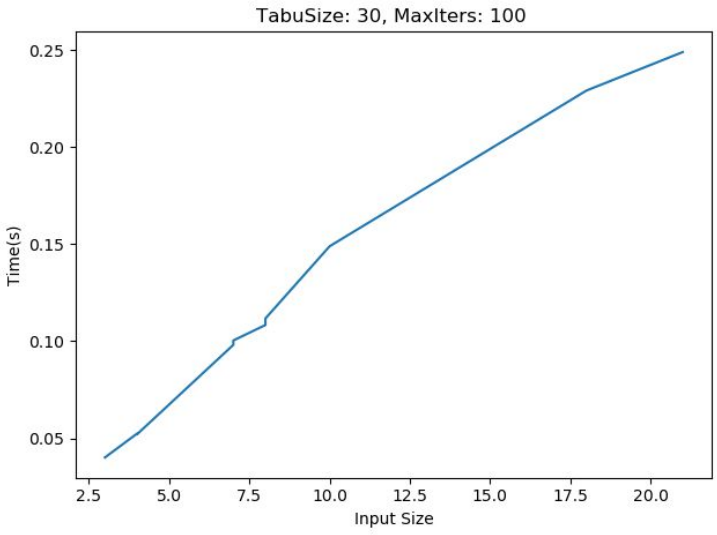
Bu bölümde proje için hazırlanan tüm test caseleri için Tabu Search test sonuçları gösterilecektir. Bu test caselerine raporun ilerleyen bölümlerinde ayrıca değinilmiştir.

Parametrelerin zamana etkisi(TabuSize: değişken, MaxIters: sabit)

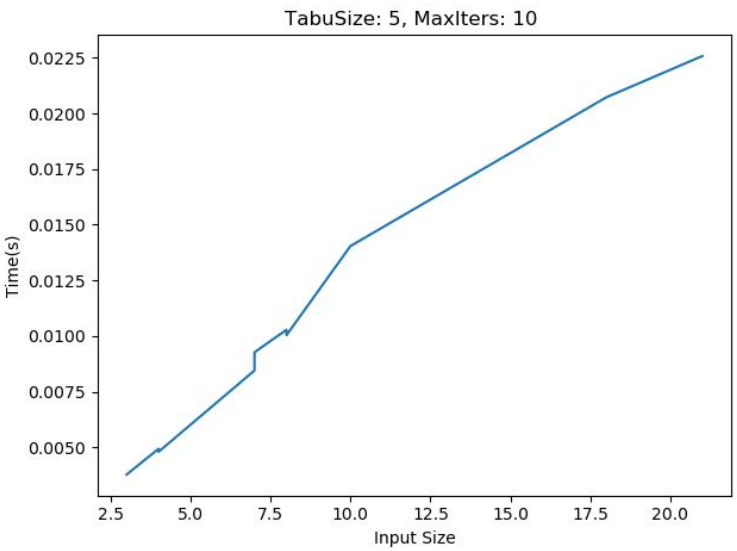


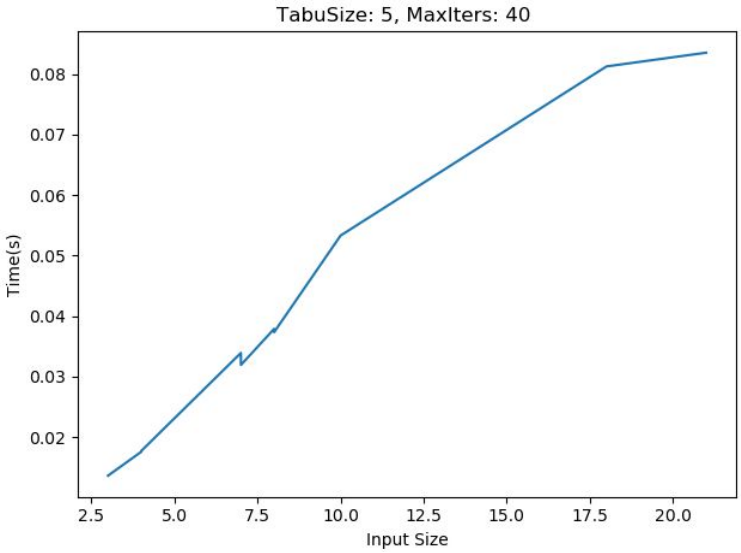


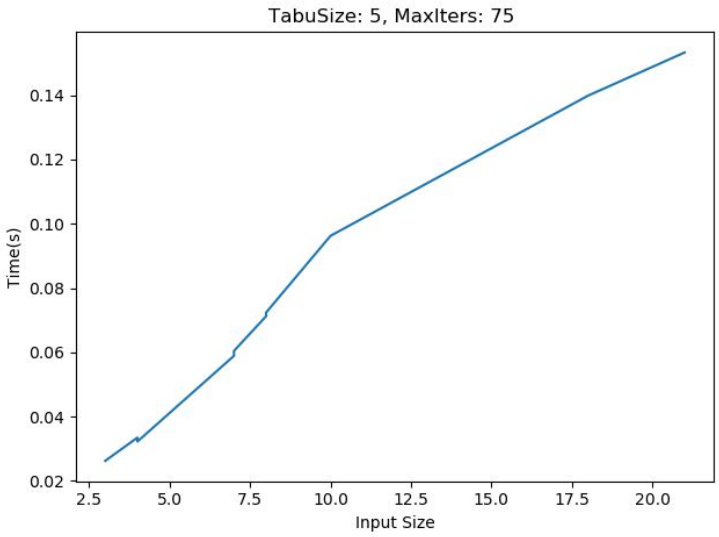


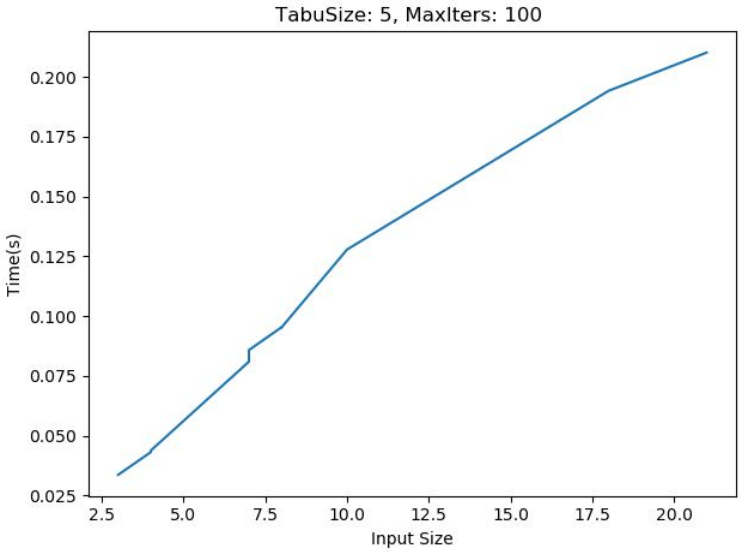


Parametrelerin zamana etkisi(TabuSize: sabit, MaxIters: değiken)









Bu testlerden zaman karmaşıklığını doğru saptandığını görebiliyoruz.

Parametrelerin cost minimizasyonuna etkisi(TabuSize: değişken, MaxIters: sabit)

	Minimized Cost (MaxIters: 100)			
#Test	TabuSize: 5	TabuSize: 10	TabuSize: 15	TabuSize: 30
Test 1	149.7124684	149.7124684	149.7124684	149.7124684
Test 2	97.97435476	97.97435476	97.97435476	97.97435476
Test 3	107.3386304	107.3386304	107.3386304	107.3386304
Test 4	247.4596669	247.4596669	247.4596669	247.4596669
Test 5	247.4596669	247.4596669	247.4596669	247.4596669
Test 6	384	384	384	384
Test 7	294.8342566	294.8342566	294.8342566	294.8342566
Test 8	580.54834	580.54834	580.54834	580.54834
Test 9	181.3212403	181.3212403	181.3212403	181.3212403
Test 10	123.6112838	123.6112838	123.6112838	123.6112838

Bu tabloda maksimum iterasyon 100 iken, tabu listesinin boyutunun cost minimizasyonunu etkilemediğini görüyoruz.

Parametrelerin cost minimizasyonuna etkisi(TabuSize: sabit, MaxIters: değişken)

	Minimized Cost (TabuSize: 5)			
#Test	maxIters: 1	maxIters: 5	maxIters: 25	maxIters: 100
Test 1	149.7124684	149.7124684	149.7124684	149.7124684
Test 2	97.97435476	97.97435476	97.97435476	97.97435476
Test 3	107.3386304	107.3386304	107.3386304	107.3386304
Test 4	262.3790008	247.4596669	247.4596669	247.4596669
Test 5	262.3790008	247.4596669	247.4596669	247.4596669
Test 6	384	384	384	384
Test 7	297.0978478	304.0318512	296.7935396	294.8342566
Test 8	580.54834	580.54834	580.54834	580.54834
Test 9	201.4689966	203.0822406	182.8700879	181.3212403
Test 10	164.9493177	164.9493177	123.6112838	123.6112838

Bu tabloda tabu sayımız 5 iken, iterasyon sayısını arttırmanın bazı testlerde daha iyi çözümler çıkardığını görebiliyoruz.

Genetic Algorithm(GA)

Strateji

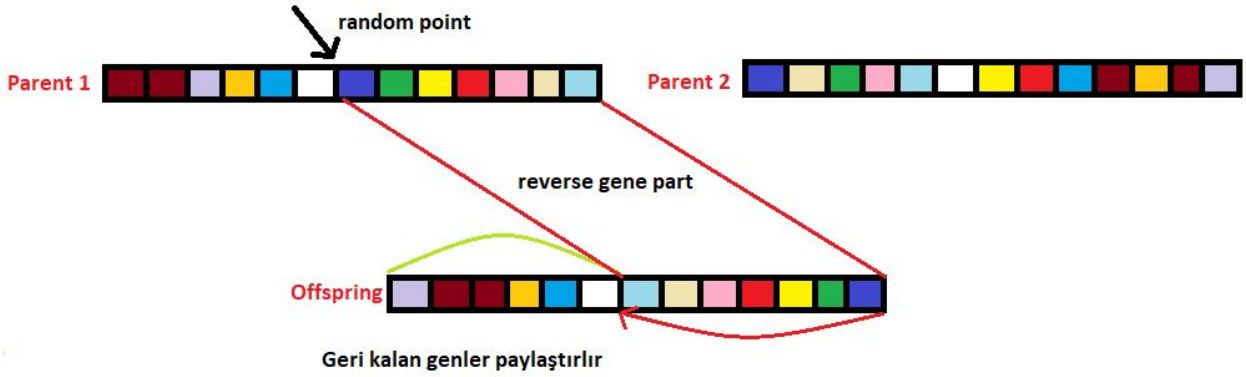
GA'da öncelikle çözümlerin(individual) nasıl temsil edileceğine karar verilir. Bu belirlendikten sonra, bu çözümlerle bir çözüm havuzu(population) oluşturulur. Her iterasyonda bu çözüm havuzuna selection, mutation ve crossover uygulanarak yeni nesil belirlenir.

Bu implementasyonda individual **gen** ve **fitness** olmak üzere iki özellik barındırmaktadır. Gen bir çözümü eder, bu çözüm verilen problemde çemberlerin bir permütasyonudur. Fitness ise bu çemberleri sığacağı genişliktir. Yani fitness değeri küçük olan bireyleri arıyoruz. Burada isimlendirmede biraz ters mantık olmuş, fitness yerine cost olarak düşünebiliriz.

Popülasyon oluşturulurken her birey rastgele bir gen atanır. Bu gen çember diziliminin bir permütasyonudur. Bu atanan gen için fitness değeri belirlenir.

Seleksiyon strateji olarak tournament selection kullanılmıştır. Turnuvaya çıkacak 4 birey belirlenir ve bu 4 bireyden en iyisi gelecek nesile devam eder.

Crossover strateji için permütasyon tabanlı çözümler için özelleştirilmiş one-point crossover kullanıldı.



Bu crossover sayesinde aslında yeni rastgele bir permütasyon üretmiş oluyoruz ve aynı zamanda ebeveynlerden de dizilim düzeninin belli bir kısmını aktarıyoruz. Birinci ebeveynden seçtiğimiz gen parçasını ters çevirirken ve geri kalan gen parçalarını çoçuk bireye dizerken de çözüm uzayında farklı yönlere gitmeye olanak tanınıyor böyle diversification sağlanıyor. Oluşan çocuk birey, popülasyondaki en güçsüz bireyin yerini alıyor

Mutasyon olarak ise iki tane rastgele gen parçasının yeri değiştirilmekte.

Seleksiyon, crossover ve mutasyon stratejilerinin uygulanması sırası ise generation recipe de belirleniyor.

```
def generationRecipe(self):
    self.selection()
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.mutation(self.population.getWeakestMember())
    self.mutation(self.population.getWeakestMember())
    self.mutation(self.population.getWeakestMember())
```

Kodda da görüldüğü üzere önce seleksiyon uygulanıyor sonra yeni oluşan popülasyon üzerinde 6 birey çiftleştiriliyor ve en güçsüz 3 bireye de mutasyon uygulanıyor. Bu tarif değiştirilebilir, herhangi bir method kaldırılabilir. Seleksiyon yöntemi intensification yaparken, crossover ve mutasyon yöntemleri diversification yapmaktadır.

Pseudocode

GA'da short-term memory yok fakat çok küçükte olsa bir long-term memory var. Bu memory bugüne kadar yaşamış en iyi bireyi tutuyor. Geleneksel olarak GA uygulamalarında böyle bir uzun süreli hafızaya rastlamadım fakat bugüne kadar bulduğumuz en iyi çözümü elimizde tutmak gayet mantıklı. Ayrıca short-term memory olmasadan her yeni jenarasyon kendinden önce gelene bir nebzede olsa benziyor, hatta bazı bireyler olduğu gibi yeni jenerasyona geçmiş olabiliyor.

$Input: \ Population_{size}, \ Generation_{size}$ //Population => Solution Space, Generation => MaxIters

$Output: \ BestIndividualEverLived$

$For \ (0 \ to \ \ Generation_{size})$

$\quad population.generationRecipe()$

$return \ population.bestIndividualEverLived$

Complexity

GA'da döngü kullanan sadece seleksiyon metodu ve GA tarifinin uygulandığı metod var. Seleksiyon metodu, popülasyon sayınca dönüyor, her adımda 1 turnuva oynanıyor ve bu turnuvanın oynanması sabit zamanlı bir işlem. GA tarifi uygulanırken de istenen jenerasyon sayısınca dönüyor. Sonuç olarak $O(pop\ x\ generation)$, yani lineer zamanda çalışıyor.

Generasyon Tarifinin Ayarlanması

Tarifin minimizasyona etkisi(Popülasyon: sabit, Generasyon: sabit)

	Minimized Cost (Population: 5, Generation 10)			
#Test	No Selection	No Crossover	No Mutation	Regular Recipe
Test 1	149.7124684	209.7124684	149.7124684	149.7124684
Test 2	97.98717738	115.8628986	97.97435476	97.97435476
Test 3	148.2693923	119.0093923	107.3386304	107.3386304
Test 4	278.4435337	324.9193338	247.4596669	247.4596669
Test 5	321.3687956	359.0824435	296.4494618	256.7315542
Test 6	474.509668	490.509668	384	384
Test 7	301.5453476	390.8840075	302.3911515	303.9965968
Test 8	580.54834	690.27417	580.54834	580.54834
Test 9	214.1138467	263.8859003	204.7789902	201.6380465
Test 10	123.6112838	233.6112838	123.6112838	128.9493177

Bu tabloda crossover metodunun GA'da ki en kritik metod olduğunu gözleniyor. Yani crossover içermeyen bir tarifte, GA çok kötü jenerasyonlara gidiyor. Seleksiyon kullanmadığımızda ise yine belli bir miktar vasat çözümlere ulaştığımız görebiliyoruz. Mutasyonda jenerasyonlar en az etkileyen faktör. Gözlemler dikkate alınarak yeni bir tarif oluşturuldu. Bu tarifte her crossoverladan sonra mutasyon yapılarak, eğer crossover sonucu oluşan offspring en kötü offspring ise mutasyona uğruyor.

Yeni tarif:

```
def generationRecipe(self):
    self.selection()
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.mutation(self.population.getWeakestMember())
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.mutation(self.population.getWeakestMember())
    self.crossover(self.population.getRandomMember(), self.population.getRandomMember())
    self.mutation(self.population.getWeakestMember())
```

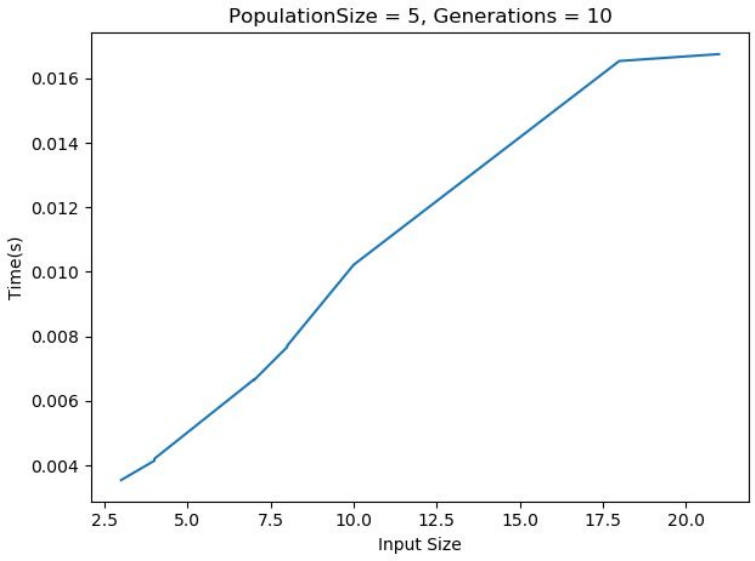
Tarifin minimizasyona etkisi(Popülasyon: sabit, Generasyon: sabit)

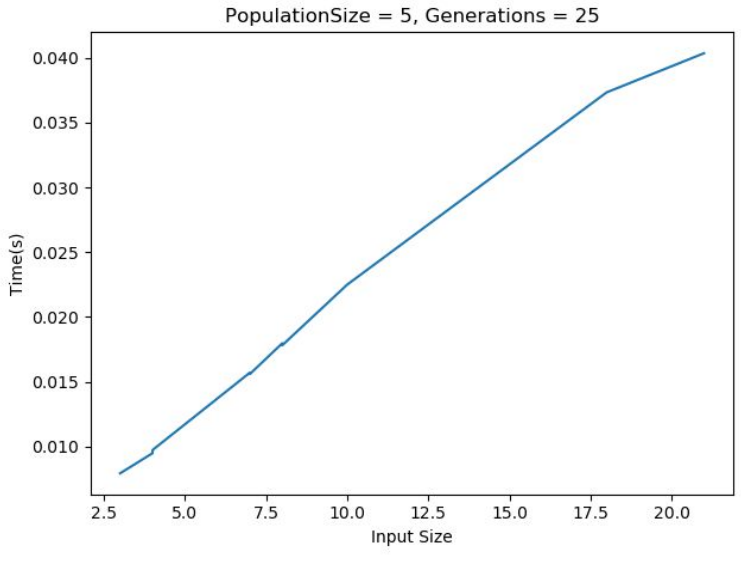
	Minimized Cost (Population: 5, Generation 10)				
#Test	No Selection	No Crossover	No Mutation	Regular Recipe	Adjusted Recipe
Test 1	149.7124684	209.7124684	149.7124684	149.7124684	149.7124684
Test 2	97.98717738	115.8628986	97.97435476	97.97435476	97.97435476
Test 3	148.2693923	119.0093923	107.3386304	107.3386304	107.3386304
Test 4	278.4435337	324.9193338	247.4596669	247.4596669	247.4596669
Test 5	321.3687956	359.0824435	296.4494618	256.7315542	247.4596669
Test 6	474.509668	490.509668	384	384	384
Test 7	301.5453476	390.8840075	302.3911515	303.9965968	299.7924065
Test 8	580.54834	690.27417	580.54834	580.54834	580.54834
Test 9	214.1138467	263.8859003	204.7789902	201.6380465	184.3090776
Test 10	123.6112838	233.6112838	123.6112838	128.9493177	124.9493177

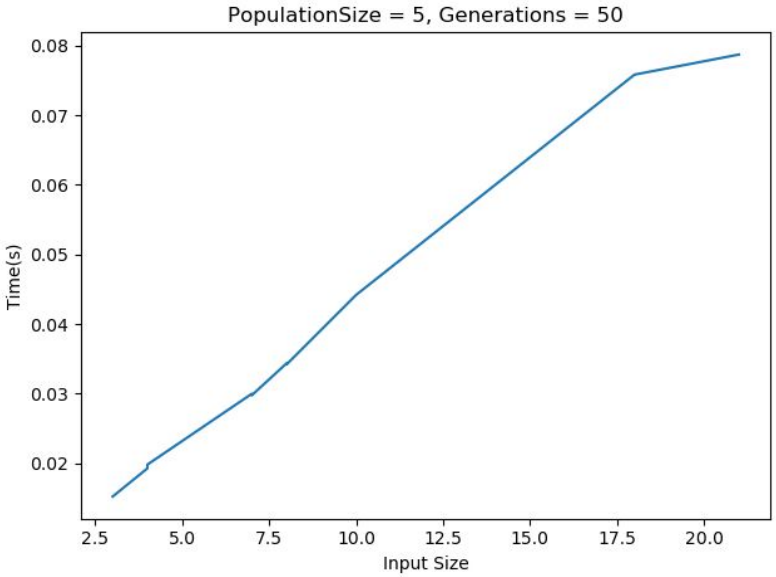
Yeni tarife göre test sonuçları tabloya eklendikten sonra, jenerasyonların çoğu testte daha iyi yerlere gittiğini görebiliyoruz. Geri kalan parametre ayarlarında da bu jenerasyon tarifi ile devam edilecektir

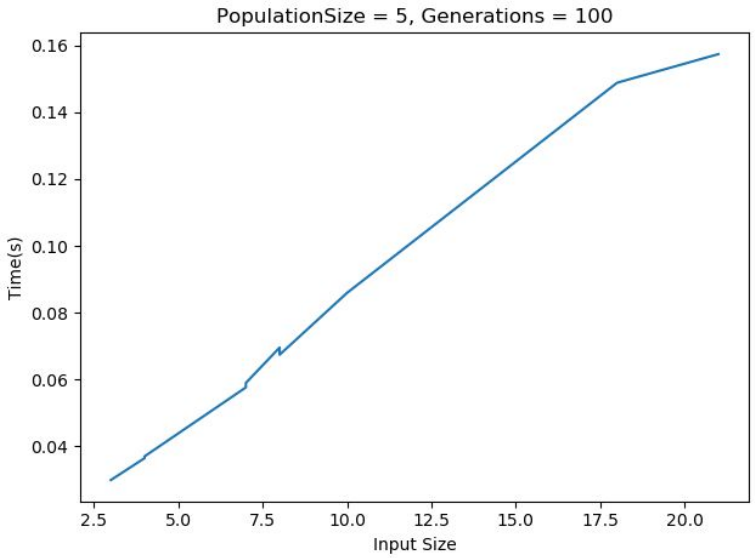
Parametre Testleri

Parametrelerin zamana etkisi(Popülasyon: sabit, Generasyon: değişken)









Buradaki grafiklerden zaman karmaşıklığının gerçekten lineer olduğunu gözlemlemiş olduk. Jenerasyon sayısını sabit tutarak tekrar test yapma gereği duymadım.

Parametrelerin minimizasyona etkisi(Popülasyon: sabit, Generasyon: değişken)

	Minimized Cost (Population: 5)		
#Test	Generation: 5	Generation: 50	Generation: 100
Test 1	149.7124684	149.7124684	149.7124684
Test 2	97.97435476	97.97435476	97.97435476
Test 3	107.3386304	107.3386304	107.3386304
Test 4	262.3790008	247.4596669	247.4596669
Test 5	247.4596669	247.4596669	247.4596669
Test 6	384	384	384
Test 7	329.0059783	297.0978478	297.8851406
Test 8	610.27417	580.54834	580.54834
Test 9	212.5591115	186.4811593	186.3620213
Test 10	164.9493177	130.2975753	123.6112838

Jenerasyon sayısının artması büyük çözüm kümelerinde da iyi sonuçlar alınmasını sağlıyor. (Test1 size: 3, Test10 size : 20)

Parametrelerin minimizasyona etkisi(Popülasyon: değişken, Generasyon: sabit)

	Minimized Cost (Generation: 25)		
#Test	Pop: 5	Pop: 10	Pop: 25
Test 1	149.7124684	149.7124684	149.7124684
Test 2	97.97435476	97.97435476	97.97435476
Test 3	107.3386304	107.3386304	107.3386304
Test 4	247.4596669	247.4596669	247.4596669
Test 5	247.4596669	247.4596669	247.4596669
Test 6	384	384	384
Test 7	294.8342566	297.0978478	297.8851406
Test 8	580.54834	580.54834	580.54834
Test 9	186.1249651	187.1711004	191.3909791
Test 10	123.6112838	154.9838668	123.6112838

Popülasyon sayımızı arttırmak en iyi çözümü fazla etkilemiyor o yüzden 5 ila 10 arası değerler popülasyon için ideal.

Particle Swarm Optimization(PSO)

Strateji

PSO'da popülasyon tabanlı bir yaklaşım olduğu için GA ile benzerlik göstermektedir. GA gibi evrimsel operatörleri yoktur fakat her birey belli bir oranda diğer bireylerden etkilenir. PSO, particle ve swarm olmak üzere iki ana parçadan oluşur. Bu implementasyonda particle üzerinde iki işlem mümkündür. Bunlardan ilki evaluate metodudur. Bu diğer yaklaşımlardaki cost metodlarıyla aynıdır. İkinci metod ise update metodudur. Basit PSO implementasyonunda pozisyonun ve hızın güncellenmesi ayrı işlemlerdir. Problemi PSO ile çözülebilir bir temsile getirebilmek adına particle güncelleme kısmı düzenlenmelidir. Elimizdeki problem belli sayıda çemberin farklı sıralamasıdır. Klasik PSO'daki hız kavramını direk olarak bu sıralı permütasyon problemine uygulamak mümkün olmadığı için update metodu PSO'nun temel mantığına sadık kalarak modifiye edilmiştir.

Pseudocode

Update metodu:

$Input : GlobalBest, w, c1, c2$

$diversify = |w * cost|$

$cognitive = |c1 * r1 * (personalBestCost - cost)|$

$social = |c2 * r2 * (globalBestCost - cost)|$

$If\ (diversify > social\ \&\ diversify > cognitive)$

>Update position with new permutation

$Else\ If\ (cognitive > social)$

>Update position with personalBestSolution

Else:

>Update position with globalBestSolution

```
31      def update(self, globalBest, globalBestCost, w= 0.5, c1 = 1.5, c2 = 1.5):
32          r1 = random.random()
33          r2 = random.random()
34
35          diversify = math.fabs(w * self.cost)
36          cognitive = math.fabs(c1 * r1 * (self.bestCost - self.cost))
37          social = math.fabs(c2 * r2 * (globalBestCost - self.cost))
38
39          countDiverse = 0
40          countCognitive = 0
41          countSocial = 0
42
43          if diversify > social and diversify > cognitive:
44              self.position = utilities.randomPermutaion(self.position)
45              self.cost = self.evaluate()
46              countDiverse += 1
47              #print("diversify")
48          elif cognitive > social:
49              self.position = self.bestPosition
50              self.cost = self.bestCost
51              countCognitive +=1
52              #print("cognitive")
53          else:
54              self.position = globalBest
55              self.cost = globalBestCost
56              countSocial += 1
57              #print("social")
58
59          return countDiverse,countCognitive,countSocial
```

PSO metodu:

```
Input : solutionSpace, swarmSize, w, c1, c2, maxIters, noImprovementIters
Output : globalBest
noImprov = 0
Swarm = list(size=swarmSize)
For(i = 0 to swarmSize)
    swarm[i] ← Particle(solutionSpace) //Random permutation of solution space
For(i = 0 to maxIters || i == noImprovementıters)
    For(particle in swarm)
        particle.evaluate()
        If ( particle.cost < globalBestCost)
            Update globalBest with particle
            noImprov = 0
        noImprov += 1
    For(particle in swarm)
        particle.update(globalBest)
    If (noImprov >= noImprovementIters)
        break
return GlobalBest
```

Update metodu yapılan 3 çeşit hesaplamaya göre particle hangi yönde gideceğine karar veriyor.

- Diversification ile sonuçlanan güncellemelerde particle tamamen yenileniyor.
- Cognitive ‘de kişisel en iyiye yöneliyor.
- Social ‘da global en iyiye yöneliyor

Complexity

Zaman karmaşıklığı açık bir şekilde lineer, O(n) ve maxIteration * (swarmSize * 2) ile doğru orantılı.

Parametrelerin Ayarlanması

Bu implementasyonda PSO'nun 5 tane hiper parametresi var.

- W: diversification çarpanı
- C1: cognition çarpanı
- C2: socialization çarpanı
- maxIter: Sürünün geçirdiği döngü sayısı
- noImprov: Yeni global best oluşmayan döngü sayısı

W değişken, diğer her şey sabit

Particle Decision (swarmSize = 10, c1 = 1.9, c2 = 1.1, maxIters = 100,noImp= 50)				
#Test	**#update result**	w:1.5	w:0.4	w:0.1
Test 1	**diversification**	500	444	203
	cognition	0	55	176
	socialization	0	1	121
Test 2	**diversification**	500	500	331
	cognition	0	0	111
	socialization	0	0	58
Test 3	**diversification**	500	484	184
	cognition	0	16	182
	socialization	0	0	134
Test 4	**diversification**	500	508	127
	cognition	0	12	197
	socialization	0	0	196
Test 5	**diversification**	510	497	163
	cognition	0	23	172
	socialization	0	0	165
Test 6	**diversification**	530	480	250
	cognition	0	20	138
	socialization	0	0	112
Test 7	**diversification**	560	989	452
	cognition	0	11	304
	socialization	0	0	244
Test 8	**diversification**	520	508	250
	cognition	0	2	145
	socialization	0	0	105
Test 9	**diversification**	590	908	348
	cognition	0	12	323
	socialization	0	0	309
Test 10	**diversification**	629	421	98
	cognition	1	70	250
	socialization	0	49	232

Yukarıdaki tabloda w değişkeni 0.1 iken update sonuçlarında düzenli bir dağılım sağlandığı gözlemleniyor. Geri kalan testler greedy, tabuSearch, GA ve PSO'nun beraber karşılaştırılması halindedir.

Ortak Testler

- TabuSearch: tabuSize: 5, maxIterations: 25
- GA: populationSize = 10, generations = 50
- PSO: swarmSize: 10, w: 0.1, c1: 1.9, c2: 1.1, maxIters: 100, stopAfterNoImprovement:50

	Cost			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1	149.7124684	149.7124684	149.7124684	149.7124684
Test 2	97.97435476	97.97435476	97.97435476	97.97435476
Test 3	107.3386304	107.3386304	107.3386304	107.3386304
Test 4	262.3790008	247.4596669	247.4596669	247.4596669
Test 5	262.3790008	247.4596669	247.4596669	247.4596669
Test 6	394.509668	384	384	384
Test 7	323.7306579	294.8342566	299.2800432	295.9258576
Test 8	642.27417	580.54834	580.54834	580.54834
Test 9	217.0688766	194.2668844	186.3620213	186.4811593
Test 10	164.9493177	123.6112838	132.9493177	123.6112838

	Time(s)			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1	0.000045	0.009002	0.019397	0.040816
Test 2	0.000069	0.012184	0.023113	0.074966
Test 3	0.000046	0.011587	0.024459	0.063767
Test 4	0.000076	0.022476	0.034566	0.099957
Test 5	0.000121	0.026553	0.038933	0.093427
Test 6	0.000087	0.030368	0.044651	0.12842
Test 7	0.000079	0.025455	0.04266	0.233177
Test 8	0.000101	0.034331	0.049827	0.17204
Test 9	0.000193	0.053357	0.086671	0.272546
Test 10	0.00017	0.056948	0.087308	0.342555

Standart testlerde PSO'nun en iyi sonuçları verdiğini fakat en yüksek zamanda çalıştığı gözlemleniyor. Belirlenen parametreler için GA'nın yaklaşık 2.5 katı, TabuSearch'ün yaklaşık 5 katı daha uzun sürede sonuçlanıyor. Her algoritmanın maksimum iterasyon sayısı karşılaştırdığında bu zaman farkının nereden geldiğini görebiliyoruz.

Büyük Rastgele Girdiler İle Testler

Bu testler için çok sayıda, rastgele oluşturulan yarıçaplar ile algoritmalar denenmiştir

Parametreler Versiyon 1

- TabuSearch: tabuSize: 25, maxIterations: 100
- GA: populationSize = 10, generations = 100
- PSO: swarmSize: 10, w: 0.1, c1: 1.9, c2: 1.1, maxIters: 100, stopAfterNoImprovement:75

	Cost			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1, size 50	4797.189969	4699.412272	4722.506021	4715.346304
Test 2, size 100	9580.813231	9392.794529	9485.083737	9493.264795
Test 3, size 250	22599.1982	22209.03156	22443.20119	22405.46675

	Time(s)			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1, size 50	0.001096	1.367167	1.4702	3.680711
Test 2, size 100	0.003132	3.830388	8.958955	9.958743
Test 3, size 250	0.014152	16.890349	104.928971	56.481369

Versiyon 1 testlerinde en iyi sonucu TabuSearch verdi, aynı zamanda en makul zamanda sonuçlandı. GA'da ki zaman artışının muhtemel nedeni list üzerinde işlemlerin en yoğun olduğu yöntem olması. GA ve PSO'da, sort ve remove kullanılmasının etkisini büyük girdilerde daha rahat görebiliyoruz.

Parametreler Versiyon 2

- TabuSearch: tabuSize: 50, maxIterations: 200
- GA: populationSize = 15, generations = 50
- PSO: swarmSize: 20, w: 0.1, c1: 1.9, c2: 1.1, maxIters: 75, stopAfterNoImprovement:100

TabuSearch, GA ve PSO'ya kıyasla çok hızlı çalıştığı için iterasyon sayısı arttırıldı. Tabu listesinin büyüklüğü de daha büyük input uzaylarında farklı yönlere gidilebilmesi için arttırıldı.

GA'da popülasyon 1.5 katında büyütülüp , jenerasyon sayısı yarıya indirildi.

PSO'da maksimum iterasyon sayısı azaltıldı, noImprov iterasyon sayısı ve swarm size artırıldı.

	Cost			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1, size 50	4565.650083	4427.190266	4586.88253	4470.871674
Test 2, size 100	8972.454797	8741.834331	8888.273452	8840.291015
Test 3, size 250	23604.01807	23195.51943	23564.10001	23470.68204

	Time(s)			
#Test	**Greedy**	**TabuSearch**	**GA**	**PSO**
Test 1, size 50	0.001	2.707573	0.758292	5.378886
Test 2, size 100	0.003038	7.781406	4.105128	16.478799
Test 3, size 250	0.013634	33.196352	56.82656	83.302207

Burada yine Tabu Search'ün en iyi sonuçları verdiği gözlemlendi. Ayrıca GA'nın ilk çözümleri rastgele oluşturulduğu için, test 1'de greedy'nin gerisinde kaldığı gözlemlendi.

Testler Nasıl Çalıştırılır

utilities.py dosyasında tests ve randomTests listeri bulunmakta. Bu testleri main.py içersinde kullanılıyor.

```
import utilities
from tabuSearch import tabuSearch
from greedy import greedy
from geneticAlgorithm import geneticAlgorithm
from pso import pso

if __name__ == "__main__":
    algorithms = [greedy, tabuSearch, geneticAlgorithm, pso]

    for i, test in enumerate(utilities.tests):
        print("#Test{}".format(i+1))
        for algorithm in algorithms:
            utilities.chronometer(algorithm, test)

```

Kırmızı altı çizili yere içerisinde testler olan bir liste göndererek istediğiniz gibi test edebilirsiniz. Test formatı utilities.py'deki gibi olmalıdır.

Parametreler Nasıl Değiştirilir

TabuSeach parametreleri, tabuSearch.py içerisinde:

```
def tabuSearch(circles, tabuSize = 5, maxIterations=25):

    current = greedy.greedy(circles)
    currentCost = cost(current)
    best = current

    tabuList = [None] * tabuSize
    iters = 0
    while(iters <= maxIterations):
        candidates = []
        for i in range(0,3):
            candidates.append(generateCandidate(best,tabuList))

        #Find lowest cost candidate
        candidates.sort(key=lambda x: cost(x), reverse=False)
        best_candidate = candidates[0]
```

GA parametreleri, geneticAlgorithm.py içersinde:

```
135  def geneticAlgorithm(circles, populationSize = 10, generations = 50):
136      formatedInput = utilities.reduceToRadius(circles)
137      ga = GeneticFramework(formatedInput, populationSize)
138
139      for i in range(0,generations):
140          ga.generationRecipe()
141
142      output = []
143
144      for r in ga.fittestIndividualLived.gene:
145          output.append([0,r])
```

PSO parametreleri, pso.py içersinde:

```
71  def pso(solutionSpace, swarmSize = 10, w = 0.1, c1 = 1.9, c2 = 1.1, maxIters = 100, stopAfterNoImprovement = 75):
72      swarm = [None] * swarmSize
73      globalBest = greedy.greedy(solutionSpace)   # Construct initial greedy solution
74      globalBestCost = cost(globalBest) # Calculate cost of the initial solution
75
76      iterNoImprovement = 0
77
78      countDiverse = 0
79      countCognitive = 0
```

Son Notlar

- Ortak testler > Ortak Testler.txt
- Parametre Versiyon 1 Testeler > v1 Testler.txt
- Parametre Versiyon 2 Testeler > v2 Testler.txt
- Testler Windows 10, Intel i5 2500 ile yapılmıştır.

Tablolar:

https://docs.google.com/spreadsheets/d/11BjMXSLqLNvqJd3UcbMmvfjs4F-Qn5vFYIE68LXZ4SE/edit?usp=sharing

Yararlanılan Kaynaklar

1. **http://www.swarmintelligence.org/tutorials.php**
2. **http://www.cleveralgorithms.com/nature-inspired/stochastic/tabu_search.html**
3. **https://nathanrooy.github.io/posts/2016-08-17/simple-particle-swarm-optimization-with-python/**
4. **Ders slaytları**
5. **Grafikler numpy ve matlotlip.pyplot kütüphaneli ile oluşturulmuştur. Gönderilen scriptler grafik çizmeyecek şekilde tekrar düzenlenip gönderilmiştir.**