Veriliş Tarihi: 12.10.2018

Son Teslim Tarihi:22.11.2018

*YAPAY ZEKA-YZM 3217*

“Yapay Zeka Tabanlı Pacman Oyunu”

HAZIRLAYANLAR:

152805010-Cennet Gizem Söylemez

152805009-Bekir Furkan Zadegil

“Yapay Zeka Tabanlı Pacman Oyunu”

İçindekiler Tablosu

PROJENİN ADI 2

PROJE KONUSU

PROJE AMACI 3

**YAZILIM GELİŞTİRME SÜREÇLERİ** 3

PLANLAMA 4

PLANLAMA AŞAMASINDA YAPILAN ÇALIŞMALAR 5

**TASARIM VE GERÇEKLEŞTİRME** **6**

PROJE KODLARI……………………………………………………………………………..7

**VERİ YAPILARI KATOLOĞU…………………………………………………………….…………………………39**

Projenin Adı : “Yapay Zeka Tabanlı Pacman Oyunu”

Projenin Konusu: Proje kapsamında gerçekleştirilen Pacman etmeninin, labirent içerisinde yol bulabilme, belirli bir konuma ulaşabilme ve etkin bir biçimde yiyecekleri toplayabilmesi amaçlanmaktadır.

Projenin Amacı: Depth first search, DFS, Breadth first search, BFS, Uniform cost search, A\* “Manhattan uzaklığını” kullanarak bu kavramları daha iyi öğrenmek .

Proje Metni: Pacman oyununun gerçekleştirimi, aşağı belirtilen arama algoritmalarının gerçekleştirilmesi ve belirtilen senaryoların uygulanması ile yapılacaktır.

1. Belirli bir sabit yiyecek kaynağına erişmek için derinlik öncelikli arama (depth first search, DFS) algoritmasın gerçekleştirini yapınız.

2. Genişlik öncelikli arama algoritmasının (breadth first search, BFS) gerçekleştirimini yapınız. PACMAN etmeninin hamlelerini genişlik öncelikli arama algoritmasına göre gerçekleştiriniz.

3. Eşit maliyetli arama (uniform cost search) algoritmasının gerçekleştirimini yapınız. PACMAN etmeninin hamlelerini eşit maliyetli arama algoritmasına göre gerçekleştiriniz.

4. A\* algoritmasının gerçekleştirimi yapınız. A\* algoritmasında sezgisel fonksiyon değerinin hesaplanması için “Manhattan uzaklığını” kullanınız. Bir başka uzaklık fonksiyonu kullanıp her ikisinin A\* algoritmasındaki etkinliğini değerlendiriniz.

5. PACMAN oyununda, her biri bir köşede olmak üzere toplam dört tane nokta bulunmaktadır. Buna göre, labirent üzerindeki dört noktaya birden değen, en kısa yolu belirlemek üzere, A\* algoritması tabanlı bir gerçekleştirim yapınız.

6. Raporunuzda, farklı algoritmaların aynı problemi çözmeye yönelik olarak etkinliklerini değerlendiriniz.

NOT: Algoritmaların gerçekleştiriminde, yığın, kuyruk, öncelik kuyruğu vb. veri yapıları kullanılabilir.

# YAZILIM GELİŞTİRME SÜREÇLERİ

* Planlama
* Çözümleme
* Tasarım Ve Gerçekleştirme

# PLANLAMA:

Yazılım geliştirme sürecinin ilk unsuru planlamadır. Proje Planının temel amacı proje sahibinin gereksinimlerini listelemektir.

# Proje Planlama Aşamasında Yapılan Çalışmalar

**1-Proje Kaynaklarının Belirlenmesi:**

**Proje Ekibinin Oluşturulması:**

Projemiz 2 kişilik grup üyelerinden

1-Cennet Gizem Söylemez

2-Bekir Furkan Zadegil

**2-Proje Zamanlaması:**

Projemiz 12 Ekim-22 Kasım tarihleri arasında yapılmıştır. Beraber belli günler belirleyip buluşulup proje bitirilmiştir.

**3-Yazılım Araçları:**

* Pycharm ile yazılmıştır.

# ÇÖZÜMLEME

Çözümleme aşaması Üretimin ilk basamağıdır.

Üretim süreci çözümleme aşamasıyla başlamıştır. Gerekli olan işlevsel veya işlevsel olmayan gereksinimler belirlenip projede üretime geçilmiştir.

# TASARIM VE GERÇEKLEŞTİRME

Yazılım tasarımı, bir sorunun yazılım çözümü için, sorun giderme ve planlama aşamasıdır. Gerçekleştirme aşamasından önce yapılan tasarım aşamasıyla gerçekleştirme aşamasının ilk girişi yapılır. Tasarım şekli belirlendikten sonra gerçekleştirilmeye başlanır.

VERİ YAPILARI KATALOĞU

Kullanılan Veri Yapıları:

* Depth first search (DFS)
* Breadth first search( BFS)
* Uniform cost search
* A\* “Manhattan uzaklığını”

# **Depth First search(dfs):**

Bir ağaç dolaşma algoritmasının (tree traverse algorithm, tree traversal) ilk önce alt seviyesinde bulunan komşularını araması durumudur.

Bilgisayar biliminde, derin öncelikli arama, ağaç ya da çizge veri yapılarında arama yapmak için kullanılan bir algoritmadır. Algoritma aramaya başladığı düğümden ulaşabileceği en derin düğüme kadar gider, gidecek daha derin bir düğüm kalmadığında geri sarar ve derin düğümlere öncelik vererek gezmeye devam eder.

**BREADTH FİRST SEARCH(BFS):**

Bilgisayar biliminde, sığ öncelikli arama ya da enine arama, bir çizgenin düğümlerini, başlangıç noktasına daha yakın olanlara öncelik vererek arayan bir algoritmadır. Algoritma ziyaret ettiği düğümlerin bütün komşularını bir kuyruğa ekler ve ziyaret edeceği düğümleri kuyruktaki sıraya göre seçer.

**UNİFORM COST SEARCH:**

Ağaçlar da bir graf örneği olduğu için algoritmanın ağaçlar üzerinde çalışması da mümkündür. Algoritma basitçe aşağıdaki şekilde tanımlanabilir:

1. Kök düğümden başla (root node)
2. En düşük maliyetli komşuya git
3. Şayet aranan düğüm bulunduysa bit, bulunmadıysa 2. Adıma geri dön

# YAZILIM GELİŞTİRME İÇİN HARCANAN GÜNLER

**Grup Elemanları Görev Paylaşımı:**

* Cennet Gizem Söylemez: 15 gün
* Bekir Furkan Zadegil:15 gün

Kodlar:

Pacman.py:

**import** sys  
**import** logging  
**import** logging.handlers  
**from** BaseHTTPServer **import** BaseHTTPRequestHandler, HTTPServer  
**from** optparse **import** OptionParser  
**from** SocketServer **import** ThreadingMixIn  
**from** threading **import** Event  
**from** threading **import** activeCount  
**import** select  
**import** socket  
**from** urlparse **import** urlparse, urlunparse  
**from** types **import** FrameType, CodeType  
**from** signal **import** signal, SIGINT  
  
**try**:  
 **import** yaml  
**except** ImportError:  
  
USAGE = **"""\  
pacman  
  
pacman [SECENEKLER] ...  
  
description: çapraz etki alanı uygulamasını çivi altına almak için bir python web proxy sunucusu  
gelişme. çapraz etki alanı için en önemli araçlardan biri  
dirilişlerden beri uygulama araç kemeri n 'bok.  
SEÇENEKLER:  
 --host=HOST proxy sunucusunu bağlamak için ana bilgisayar adı.  
 --port=PORT proxy sunucusunu bağlamak için bağlantı noktası.  
 --max-threads Maksimum aktif iplik sayısı.  
 --log-level çıkış seviyesini kaydetme  
 --proxy-config proxy yapılandırması yaml dosyası.  
"""**CONFIG\_PATH = **"pacman.yaml"***# pylint #E501 disable*LOG\_FMT = **"[%(asctime)-12s] %(levelname)-8s {%(name)s \  
%(threadName)s} %(message)s "  
  
  
def** load\_proxy\_config(path):  
  
 **with** open(path, **'r'**) **as** f:  
 result = yaml.safe\_load(f)  
 f.close()  
 **return** result  
  
  
**class** ProxyHandler(BaseHTTPRequestHandler):  
  
 **def** \_\_init\_\_(self, \*args, \*\*kwargs):  
 BaseHTTPRequestHandler.\_\_init\_\_(self, \*args, \*\*kwargs)  
 self.protocol = **"HTTP/1.0"** self.protocol\_scheme = **'http'** self.\_proxy\_config = **None** self.\_proxy\_rules = **None** self.\_pacfile\_config = **None** @property  
 **def** proxy\_config(self):  
 **if not** hasattr(self, **'\_proxy\_config'**) **or** self.\_proxy\_config **is None**:  
 self.\_proxy\_config = load\_proxy\_config(CONFIG\_PATH)  
 **return** self.\_proxy\_config  
  
 @property  
 **def** proxy\_rules(self):  
 **if not** hasattr(self, **'\_proxy\_rules'**) **or** self.\_proxy\_rules **is None**:  
 self.\_proxy\_rules = self.proxy\_config.get(**'proxy\_rules'**)  
 **return** self.\_proxy\_rules  
  
 @property  
 **def** pacfile\_config(self):  
 **if not** hasattr(self, **'\_pacfile\_config'**) **or** self.\_pacfile\_config **is None**:  
 self.\_pacfile\_config = self.proxy\_config.get(**'pacfile'**)  
 **return** self.\_pacfile\_config  
  
  
 **def** handle(self):  
 *"""  
 tüm istekleri ele alır  
 """* (ip, port) = self.client\_address  
 self.server.logger.info(**"handling request from ip: {}"**.format(ip))  
  
 **if** hasattr(self, **'allowed\_clients'**) **and** ip **not in** self.allowed\_clients:  
 self.raw\_requestline = self.rfile.readline()  
 **if** self.parse\_request():  
 self.send\_error(403)  
  
 **else**:  
 **return** BaseHTTPRequestHandler.handle(self)  
  
  
 **def** render\_pacfile(self):  
 *"""  
Proxy pacfile, proxy yapılandırmasında tanımlardan oluşur.  
 """* proxy\_rule = **"""  
 if (shExpMatch(url, "\*{name}\*"))  
 return "PROXY {forward\_host}:{forward\_port}";  
 """** proxy\_rules = [proxy\_rule.format(\*\*host) **for** host **in** self.proxy\_rules]  
  
 body = **"""  
 fonksiyon FindProxyForURL(url, host) {  
 {}  
 return "DIRECT";  
 }  
 """**.format(**''**.join(proxy\_rules))  
  
 self.wfile.write(body)  
  
  
 **def** urlparse(self, scheme, host, path, params, query, fragment):  
 *"""  
 urlparse () 'nin sonucunu bir dict olarak ayrıştırır.  
 """* **return** {  
 **'scheme'**: scheme,  
 **'host'**: host,  
 **'path'**: path,  
 **'params'**: params,  
 **'query'**: query,  
 **'fragment'**: fragment}  
  
  
 **def** urlunparse(self, request):  
 urlunparse(  
 (**''**, **''**, request[**'path'**], request[**'params'**], request[**'query'**], **''**))  
  
  
 **def** do\_GET(self):  
 *"""  
 bir GET isteği işleyebilir  
 """* request = self.urlparse(\*urlparse(self.path, self.protocol\_scheme))  
  
 **if** self.is\_pacfile\_path():  
 **return** self.render\_pacfile()  
  
 **if** request[**'fragment'**]:  
 self.send\_error(400, **"unsupported url: {}"**.format(self.path))  
 **return  
  
 if** request[**'scheme'**] != **'http'**:  
 self.send\_error(400, **"unsupported url: {}"**.format(self.path))  
 **return  
  
 if not** request[**'host'**]:  
 self.send\_error(400, **"bad url: {}"**.format(self.path))  
 **return** self.send\_proxy\_request(request, self.proxy\_rules[request[**'host'**]])  
  
  
 **def** is\_pacfile\_path(self):  
 *"""  
 Geçerli istek yolu pacfile ise bir boolean döndürür.  
 """* **return** self.path.lower() == self.pacfile\_config[**'path'**].lower()  
  
  
 **def** send\_proxy\_request(self, request, forward\_proxy):  
 *"""  
 vekil bir istekte bulunur.  
 """* body = **''** length = self.headers.get(**"Content-Length"**, 0)  
 **if** length > 0:  
 body = self.rfile.read(int(length))  
 self.server.logger.debug(  
 **"(content-length {}): "**.format(length, body))  
  
 proxy\_s = self.create\_proxy\_socket()  
 **try**:  
 **if** self.proxy\_socket\_connect(proxy\_s, forward\_proxy.port):  
 self.log\_request()  
  
 proxy\_s.send(**"{} {} {}\r\n"**.format(  
 self.command, self.urlunparse(request), self.protocol))  
  
 self.headers[**'Connection'**] = **'close'  
 del** self.headers[**'Proxy-Connection'**]  
 self.server.logger.debug(**'URL: {} request'**.format(self.path))  
  
 [proxy\_s.send(**"{}: {}\r\n"**.format(\*kv)) **for** kv **in** self.headers.items()]  
  
 proxy\_s.send(**"\r\n{}\r\n"**.format(body))  
 self.socket\_rw(proxy\_s)  
 **except** Exception, e:  
 self.server.logger.exception(e)  
  
 **finally**:  
 proxy\_s.close()  
 self.connection.close()  
  
  
 **def** create\_proxy\_socket(self):  
 **return** socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
  
 **def** proxy\_socket\_connect(self, proxy\_s, port):  
 self.server.logger.debug(**"connect to {}:{}"**.format(proxy\_s, port))  
 **try**:  
 proxy\_s.connect(port)  
  
 **except** socket.error, arg;  
 self.server.logger.exception(e)  
 **try**:  
 msg = arg[1]  
 **except**:  
 msg = arg  
 self.send\_error(404, msg)  
 **return** 0  
  
 **return** 1  
  
  
 **def** socket\_rw(self, proxy\_s, max\_idle\_ticks=20):  
 iw = [self.connection, proxy\_s]  
 ow = []  
 timer = 0  
  
 **while** 1:  
 timer += 1  
 (ins, \_, exs) = select.select(iw, ow, iw, 1)  
 **if** exs:  
 **break  
  
 if** ins:  
 **for** i **in** ins:  
 **if** i **is** proxy\_s:  
 out = self.connection  
 **else**:  
 out = proxy\_s  
  
 data = i.recv(8192)  
 **if** data:  
 out.send(data)  
 timer = 0  
  
 **if** timer == max\_idle\_ticks:  
 **break** do\_PUT = do\_GET  
 do\_HEAD = do\_GET  
 do\_POST = do\_GET  
 do\_DELETE = do\_GET  
 do\_OPTIONS = do\_GET  
  
  
**class** ProxyHTTPServer(ThreadingMixIn, HTTPServer):  
 **def** \_\_init\_\_(self, server\_address, request\_handler\_class):  
 HTTPServer.\_\_init\_\_(self, server\_address, request\_handler\_class)  
 self.logger = self.create\_logger()  
  
 **def** create\_logger(self):  
 rv = logging.getLogger(self.\_\_class\_\_.\_\_name\_\_.split(**'.'**)[-1])  
 rv.setLevel(logging.INFO)  
 handler = logging.StreamHandler()  
 formatter = logging.Formatter(LOG\_FMT)  
 handler.setFormatter(formatter)  
 rv.addHandler(handler)  
 **return** rv  
  
  
**class** ProxyContext(object):  
 **def** \_\_init\_\_(self, host, port, max\_threads):  
  
 self.host = host  
 self.port = port  
 self.max\_threads = max\_threads  
 self.active\_thread\_count = 0  
 self.handler\_class = ProxyHandler  
 self.set\_sigint\_handler()  
  
  
 @property  
 **def** hostname(self):  
 **return** (self.host, self.port)  
  
  
 **def** run(self):  
 self.server = ProxyHTTPServer(self.hostname, self.handler\_class)  
  
 host, port = self.server.socket.getsockname()[:2]  
 self.server.logger.info(  
 **"serving http on host: {} port: {}"**.format(host, port))  
  
 self.active\_thread\_count = 0  
  
 **while not** self.sigint\_event.isSet():  
 **try**:  
 self.server.handle\_request()  
 self.active\_thread\_count = activeCount()  
 self.server.logger.info(  
 **"active thread count: {}"**.format(self.active\_thread\_count))  
  
 **if** self.active\_thread\_count >= self.max\_threads:  
 self.on\_max\_threads()  
  
 **except** select.error, err:  
 self.server.logger.debug(**"select.error: {}"**.format(e))  
 self.on\_sigint(\*err)  
  
 self.server.logger.info(**"proxy server shutdown.."**)  
  
  
 **def** on\_max\_threads(self):  
 self.server.logger.warn(  
 **"maksimum etkin iş parçacığına ulaşıldı. sayımı sıfırla .."**)  
 *#* ***todo: Buraya gelmek için daha fazla eylem?*** self.active\_thread\_count = 0  
  
  
 **def** set\_sigint\_handler(self):  
 *"""  
 'KeyboardInterrupt' olaylarını dinlemek için bir sinyali bağlar.  
 """* self.sigint\_event = Event()  
 signal(SIGINT, self.sigint\_handler)  
  
  
 **def** sigint\_handler(self, sigint, frame):  
 *"""  
 KeyboardInterrupt etkinlikleri için olay işleyici yöntemi.  
 """* self.server.logger.debug(  
 **"SIGINT event received: {}, {}"**.format(sigint, frame))  
  
 **while** frame **and** isinstance(frame, FrameType):  
 **if** frame.f\_code **and** isinstance(frame.f\_code, CodeType):  
 *# bu ProxyContext örneğini bulmak için yığından gider* **if "self" in** frame.f\_code.co\_varnames:  
 \_self = frame.f\_locals[**"self"**]  
 **if** isinstance(\_self, ProxyContext):  
 \_self.sigint\_event.set()  
 exit(0)  
 frame = frame.f\_back  
  
  
 **def** on\_sigint(self, code, msg, \*args):  
  
 self.server.logger.debug(**"on\_sigint, kodu: {} msg: {}"**.format(code, msg))  
  
 **if** code != 4 **and not** self.sigint\_event.isSet():  
 self.server.logger.critical(**"on\_sigint hatası, kodu: {}, {}"**.format(code, msg))  
 self.fail(msg)  
  
  
**def** fail(message, \*args):  
 print >> sys.stderr, **'error:'**, message.format(\*args),  
 exit(1)  
  
  
**def** usage():  
 print >> sys.stdout, USAGE,  
  
  
parser = OptionParser(  
 description=**"""description: çapraz etki alanı oluşturmak için bir python web proxy sunucusu  
uygulama geliştirme."""**,  
 prog=**"pacman"**,  
 usage=**"%prog [OPTIONS]"**)  
  
parser.add\_option(  
 **"--host"**, dest=**"host"**, default=**"localhost"**,  
 help=**"proxy sunucusunu bağlamak için ana bilgisayar adı."**)  
  
parser.add\_option(  
 **"--port"**, dest=**"port"**, default=3128,  
 help=**"proxy sunucusunu bağlamak için bağlantı noktası."**)  
  
parser.add\_option(  
 **"--max-threads"**, dest=**"max\_threads"**, default=1000,  
 help=**"aktif ipliklerin maksimum sayısı."**)  
  
parser.add\_option(  
 **"--proxy-config"**, dest=**"proxy\_config"**, default=**"pacman.yaml"**,  
 help=**"proxy pacfile yapılandırması dosyası."**)  
  
parser.add\_option(  
 **"--log-level"**, dest=**"log\_level"**, metavar=**"LEVEL"**,  
 help=**"çıkış seviyesi kaydediliyor."**)  
  
  
**def** main():  
 (options, args) = parser.parse\_args()  
  
 ProxyContext(  
 options.host, options.port, options.max\_threads).run()  
 exit(0)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

game.py:

**from** util **import** \*  
**from** util **import** raiseNotDefined  
**import** time, os  
**import** traceback  
  
**try**:  
 **import** boinc  
 \_BOINC\_ENABLED = **True  
except**:  
 \_BOINC\_ENABLED = **False  
  
class** Agent:  
  
 **def** \_\_init\_\_(self, index=0):  
 self.index = index  
  
 **def** getAction(self, state):  
  
 raiseNotDefined()  
  
**class** Directions:  
 NORTH = **'North'** SOUTH = **'South'** EAST = **'East'** WEST = **'West'** STOP = **'Stop'** LEFT = {NORTH: WEST,  
 SOUTH: EAST,  
 EAST: NORTH,  
 WEST: SOUTH,  
 STOP: STOP}  
  
 RIGHT = dict([(y,x) **for** x, y **in** LEFT.items()])  
  
 REVERSE = {NORTH: SOUTH,  
 SOUTH: NORTH,  
 EAST: WEST,  
 WEST: EAST,  
 STOP: STOP}  
  
**class** Configuration:  
  
 **def** \_\_init\_\_(self, pos, direction):  
 self.pos = pos  
 self.direction = direction  
  
 **def** getPosition(self):  
 **return** (self.pos)  
  
 **def** getDirection(self):  
 **return** self.direction  
  
 **def** isInteger(self):  
 x,y = self.pos  
 **return** x == int(x) **and** y == int(y)  
  
 **def** \_\_eq\_\_(self, other):  
 **if** other == **None**: **return False  
 return** (self.pos == other.pos **and** self.direction == other.direction)  
  
 **def** \_\_hash\_\_(self):  
 x = hash(self.pos)  
 y = hash(self.direction)  
 **return** hash(x + 13 \* y)  
  
 **def** \_\_str\_\_(self):  
 **return "(x,y)="**+str(self.pos)+**", "**+str(self.direction)  
  
 **def** generateSuccessor(self, vector):  
  
 x, y= self.pos  
 dx, dy = vector  
 direction = Actions.vectorToDirection(vector)  
 **if** direction == Directions.STOP:  
 direction = self.direction *# There is no stop direction* **return** Configuration((x + dx, y+dy), direction)  
  
**class** AgentState:  
 *"""  
 AgentStates bir ajanın durumunu (yapılandırma, hız, korkmuş, vb.) Tutar.  
 """* **def** \_\_init\_\_( self, startConfiguration, isPacman ):  
 self.start = startConfiguration  
 self.configuration = startConfiguration  
 self.isPacman = isPacman  
 self.scaredTimer = 0  
  
 **def** \_\_str\_\_( self ):  
 **if** self.isPacman:  
 **return "Pacman: "** + str( self.configuration )  
 **else**:  
 **return "Ghost: "** + str( self.configuration )  
  
 **def** \_\_eq\_\_( self, other ):  
 **if** other == **None**:  
 **return False  
 return** self.configuration == other.configuration **and** self.scaredTimer == other.scaredTimer  
  
 **def** \_\_hash\_\_(self):  
 **return** hash(hash(self.configuration) + 13 \* hash(self.scaredTimer))  
  
 **def** copy( self ):  
 state = AgentState( self.start, self.isPacman )  
 state.configuration = self.configuration  
 state.scaredTimer = self.scaredTimer  
 **return** state  
  
 **def** getPosition(self):  
 **if** self.configuration == **None**: **return None  
 return** self.configuration.getPosition()  
  
 **def** getDirection(self):  
 **return** self.configuration.getDirection()  
  
**class** Grid:  
  
 **def** \_\_init\_\_(self, width, height, initialValue=**False**, bitRepresentation=**None**):  
 **if** initialValue **not in** [**False**, **True**]: **raise** Exception(**'Grids can only contain booleans'**)  
 self.CELLS\_PER\_INT = 30  
  
 self.width = width  
 self.height = height  
 self.data = [[initialValue **for** y **in** range(height)] **for** x **in** range(width)]  
 **if** bitRepresentation:  
 self.\_unpackBits(bitRepresentation)  
  
 **def** \_\_getitem\_\_(self, i):  
 **return** self.data[i]  
  
 **def** \_\_setitem\_\_(self, key, item):  
 self.data[key] = item  
  
 **def** \_\_str\_\_(self):  
 out = [[str(self.data[x][y])[0] **for** x **in** range(self.width)] **for** y **in** range(self.height)]  
 out.reverse()  
 **return '\n'**.join([**''**.join(x) **for** x **in** out])  
  
 **def** \_\_eq\_\_(self, other):  
 **if** other == **None**: **return False  
 return** self.data == other.data  
  
 **def** \_\_hash\_\_(self):  
 base = 1  
 h = 0  
 **for** l **in** self.data:  
 **for** i **in** l:  
 **if** i:  
 h += base  
 base \*= 2  
 **return** hash(h)  
  
 **def** copy(self):  
 g = Grid(self.width, self.height)  
 g.data = [x[:] **for** x **in** self.data]  
 **return** g  
  
 **def** deepCopy(self):  
 **return** self.copy()  
  
 **def** shallowCopy(self):  
 g = Grid(self.width, self.height)  
 g.data = self.data  
 **return** g  
  
 **def** count(self, item =**True** ):  
 **return** sum([x.count(item) **for** x **in** self.data])  
  
 **def** asList(self, key = **True**):  
 list = []  
 **for** x **in** range(self.width):  
 **for** y **in** range(self.height):  
 **if** self[x][y] == key: list.append( (x,y) )  
 **return** list  
  
 **def** packBits(self):  
  
 bits = [self.width, self.height]  
 currentInt = 0  
 **for** i **in** range(self.height \* self.width):  
 bit = self.CELLS\_PER\_INT - (i % self.CELLS\_PER\_INT) - 1  
 x, y = self.\_cellIndexToPosition(i)  
 **if** self[x][y]:  
 currentInt += 2 \*\* bit  
 **if** (i + 1) % self.CELLS\_PER\_INT == 0:  
 bits.append(currentInt)  
 currentInt = 0  
 bits.append(currentInt)  
 **return** tuple(bits)  
  
 **def** \_cellIndexToPosition(self, index):  
 x = index / self.height  
 y = index % self.height  
 **return** x, y  
  
 **def** \_unpackBits(self, bits):  
  
 cell = 0  
 **for** packed **in** bits:  
 **for** bit **in** self.\_unpackInt(packed, self.CELLS\_PER\_INT):  
 **if** cell == self.width \* self.height: **break** x, y = self.\_cellIndexToPosition(cell)  
 self[x][y] = bit  
 cell += 1  
  
 **def** \_unpackInt(self, packed, size):  
 bools = []  
 **if** packed < 0: **raise** ValueError, **"pozitif bir tam sayı olmalı"  
 for** i **in** range(size):  
 n = 2 \*\* (self.CELLS\_PER\_INT - i - 1)  
 **if** packed >= n:  
 bools.append(**True**)  
 packed -= n  
 **else**:  
 bools.append(**False**)  
 **return** bools  
  
**def** reconstituteGrid(bitRep):  
 **if** type(bitRep) **is not** type((1,2)):  
 **return** bitRep  
 width, height = bitRep[:2]  
 **return** Grid(width, height, bitRepresentation= bitRep[2:])  
  
**class** Actions:  
  
 *# Talimatlar* \_directions = {Directions.NORTH: (0, 1),  
 Directions.SOUTH: (0, -1),  
 Directions.EAST: (1, 0),  
 Directions.WEST: (-1, 0),  
 Directions.STOP: (0, 0)}  
  
 \_directionsAsList = \_directions.items()  
  
 TOLERANCE = .001  
  
 **def** reverseDirection(action):  
 **if** action == Directions.NORTH:  
 **return** Directions.SOUTH  
 **if** action == Directions.SOUTH:  
 **return** Directions.NORTH  
 **if** action == Directions.EAST:  
 **return** Directions.WEST  
 **if** action == Directions.WEST:  
 **return** Directions.EAST  
 **return** action  
 reverseDirection = staticmethod(reverseDirection)  
  
 **def** vectorToDirection(vector):  
 dx, dy = vector  
 **if** dy > 0:  
 **return** Directions.NORTH  
 **if** dy < 0:  
 **return** Directions.SOUTH  
 **if** dx < 0:  
 **return** Directions.WEST  
 **if** dx > 0:  
 **return** Directions.EAST  
 **return** Directions.STOP  
 vectorToDirection = staticmethod(vectorToDirection)  
  
 **def** directionToVector(direction, speed = 1.0):  
 dx, dy = Actions.\_directions[direction]  
 **return** (dx \* speed, dy \* speed)  
 directionToVector = staticmethod(directionToVector)  
  
 **def** getPossibleActions(config, walls):  
 possible = []  
 x, y = config.pos  
 x\_int, y\_int = int(x + 0.5), int(y + 0.5)  
  
 *# Kılavuz noktaları arasında, tüm aracılar düz devam etmelidir* **if** (abs(x - x\_int) + abs(y - y\_int) > Actions.TOLERANCE):  
 **return** [config.getDirection()]  
  
 **for** dir, vec **in** Actions.\_directionsAsList:  
 dx, dy = vec  
 next\_y = y\_int + dy  
 next\_x = x\_int + dx  
 **if not** walls[next\_x][next\_y]: possible.append(dir)  
  
 **return** possible  
  
 getPossibleActions = staticmethod(getPossibleActions)  
  
 **def** getLegalNeighbors(position, walls):  
 x,y = position  
 x\_int, y\_int = int(x + 0.5), int(y + 0.5)  
 neighbors = []  
 **for** dir, vec **in** Actions.\_directionsAsList:  
 dx, dy = vec  
 next\_x = x\_int + dx  
 **if** next\_x < 0 **or** next\_x == walls.width: **continue** next\_y = y\_int + dy  
 **if** next\_y < 0 **or** next\_y == walls.height: **continue  
 if not** walls[next\_x][next\_y]: neighbors.append((next\_x, next\_y))  
 **return** neighbors  
 getLegalNeighbors = staticmethod(getLegalNeighbors)  
  
 **def** getSuccessor(position, action):  
 dx, dy = Actions.directionToVector(action)  
 x, y = position  
 **return** (x + dx, y + dy)  
 getSuccessor = staticmethod(getSuccessor)  
  
**class** GameStateData:  
  
 **def** \_\_init\_\_( self, prevState = **None** ):  
 *"""  
Bir öncekinden bilgileri kopyalayarak yeni bir veri paketi oluşturur.  
 """* **if** prevState != **None**:  
 self.food = prevState.food.shallowCopy()  
 self.capsules = prevState.capsules[:]  
 self.agentStates = self.copyAgentStates( prevState.agentStates )  
 self.layout = prevState.layout  
 self.\_eaten = prevState.\_eaten  
 self.score = prevState.score  
 self.\_foodEaten = **None** self.\_capsuleEaten = **None** self.\_agentMoved = **None** self.\_lose = **False** self.\_win = **False** self.scoreChange = 0  
  
 **def** deepCopy( self ):  
 state = GameStateData( self )  
 state.food = self.food.deepCopy()  
 state.layout = self.layout.deepCopy()  
 state.\_agentMoved = self.\_agentMoved  
 state.\_foodEaten = self.\_foodEaten  
 state.\_capsuleEaten = self.\_capsuleEaten  
 **return** state  
  
 **def** copyAgentStates( self, agentStates ):  
 copiedStates = []  
 **for** agentState **in** agentStates:  
 copiedStates.append( agentState.copy() )  
 **return** copiedStates  
  
 **def** \_\_eq\_\_( self, other ):  
 *"""  
 Karşılaştırılacak iki durum sağlar.  
 """* **if** other == **None**: **return False** *#* ***TODO Check for type of other* if not** self.agentStates == other.agentStates: **return False  
 if not** self.food == other.food: **return False  
 if not** self.capsules == other.capsules: **return False  
 if not** self.score == other.score: **return False  
 return True  
  
 def** \_\_hash\_\_( self ):  
 **for** i, state **in** enumerate( self.agentStates ):  
 **try**:  
 int(hash(state))  
 **except** TypeError, e:  
 print e  
 *#hash(state)* **return** int((hash(tuple(self.agentStates)) + 13\*hash(self.food) + 113\* hash(tuple(self.capsules)) + 7 \* hash(self.score)) % 1048575 )  
  
 **def** \_\_str\_\_( self ):  
 width, height = self.layout.width, self.layout.height  
 map = Grid(width, height)  
 **if** type(self.food) == type((1,2)):  
 self.food = reconstituteGrid(self.food)  
 **for** x **in** range(width):  
 **for** y **in** range(height):  
 food, walls = self.food, self.layout.walls  
 map[x][y] = self.\_foodWallStr(food[x][y], walls[x][y])  
  
 **for** agentState **in** self.agentStates:  
 **if** agentState == **None**: **continue  
 if** agentState.configuration == **None**: **continue** x,y = [int( i ) **for** i **in** nearestPoint( agentState.configuration.pos )]  
 agent\_dir = agentState.configuration.direction  
 **if** agentState.isPacman:  
 map[x][y] = self.\_pacStr( agent\_dir )  
 **else**:  
 map[x][y] = self.\_ghostStr( agent\_dir )  
  
 **for** x, y **in** self.capsules:  
 map[x][y] = **'o'  
  
 return** str(map) + (**"\nScore: %d\n"** % self.score)  
  
 **def** \_foodWallStr( self, hasFood, hasWall ):  
 **if** hasFood:  
 **return '.'  
 elif** hasWall:  
 **return '%'  
 else**:  
 **return ' '  
  
 def** \_pacStr( self, dir ):  
 **if** dir == Directions.NORTH:  
 **return 'v'  
 if** dir == Directions.SOUTH:  
 **return '^'  
 if** dir == Directions.WEST:  
 **return '>'  
 return '<'  
  
 def** \_ghostStr( self, dir ):  
 **return 'G'  
 if** dir == Directions.NORTH:  
 **return 'M'  
 if** dir == Directions.SOUTH:  
 **return 'W'  
 if** dir == Directions.WEST:  
 **return '3'  
 return 'E'  
  
 def** initialize( self, layout, numGhostAgents ):  
 *"""  
Bir düzen dizisinden ilk oyun durumunu oluşturur  
 """* self.food = layout.food.copy()  
 self.capsules = layout.capsules[:]  
 self.layout = layout  
 self.score = 0  
 self.scoreChange = 0  
  
 self.agentStates = []  
 numGhosts = 0  
 **for** isPacman, pos **in** layout.agentPositions:  
 **if not** isPacman:  
 **if** numGhosts == numGhostAgents: **continue** *# Max hayaletler zaten ulaştı* **else**: numGhosts += 1  
 self.agentStates.append( AgentState( Configuration( pos, Directions.STOP), isPacman) )  
 self.\_eaten = [**False for** a **in** self.agentStates]  
  
**class** Game:  
 *"""  
Oyun, kontrol akışını yönetir, eylemleri ajanlardan talep eder.  
 """* **def** \_\_init\_\_( self, agents, display, rules, startingIndex=0, muteAgents=**False**, catchExceptions=**False** ):  
 self.agentCrashed = **False** self.agents = agents  
 self.display = display  
 self.rules = rules  
 self.startingIndex = startingIndex  
 self.gameOver = **False** self.muteAgents = muteAgents  
 self.catchExceptions = catchExceptions  
 self.moveHistory = []  
 self.totalAgentTimes = [0 **for** agent **in** agents]  
 self.totalAgentTimeWarnings = [0 **for** agent **in** agents]  
 self.agentTimeout = **False  
  
 def** getProgress(self):  
 **if** self.gameOver:  
 **return** 1.0  
 **else**:  
 **return** self.rules.getProgress(self)  
  
 **def** \_agentCrash( self, agentIndex, quiet=**False**):  
 **if not** quiet: traceback.print\_exc()  
 self.gameOver = **True** self.agentCrashed = **True** self.rules.agentCrash(self, agentIndex)  
  
 OLD\_STDOUT = **None** OLD\_STDERR = **None  
  
 def** mute(self):  
 **if not** self.muteAgents: **return  
 global** OLD\_STDOUT, OLD\_STDERR  
 **import** cStringIO  
 OLD\_STDOUT = sys.stdout  
 OLD\_STDERR = sys.stderr  
 sys.stdout = cStringIO.StringIO()  
 sys.stderr = cStringIO.StringIO()  
  
 **def** unmute(self):  
 **if not** self.muteAgents: **return  
 global** OLD\_STDOUT, OLD\_STDERR  
 sys.stdout.close()  
 sys.stderr.close()  
 sys.stdout = OLD\_STDOUT  
 sys.stderr = OLD\_STDERR  
  
  
 **def** run( self ):  
 *"""  
 Oyun oynamak için ana kontrol döngüsü.  
 """* self.display.initialize(self.state.data)  
 self.numMoves = 0  
  
 *# oyunun başlangıç ​​öğrenicilerini bilgilendir* **for** i **in** range(len(self.agents)):  
 agent = self.agents[i]  
 **if not** agent:  
 *# Bu boş bir ajan, yüklenemedi  
 # diğer takım kazanır* self.\_agentCrash(i, quiet=**True**)  
 **return  
 if** (**"registerInitialState" in** dir(agent)):  
 self.mute()  
 **if** self.catchExceptions:  
 **try**:  
 timed\_func = TimeoutFunction(agent.registerInitialState, int(self.rules.getMaxStartupTime(i)))  
 **try**:  
 start\_time = time.time()  
 timed\_func(self.state.deepCopy())  
 time\_taken = time.time() - start\_time  
 self.totalAgentTimes[i] += time\_taken  
 **except** TimeoutFunctionException:  
 print **"Ajan %d başlangıçta zaman tükendi!"** % i  
 self.unmute()  
 self.agentTimeout = **True** self.\_agentCrash(i, quiet=**True**)  
 **return  
 except** Exception,data:  
 self.unmute()  
 self.\_agentCrash(i, quiet=**True**)  
 **return  
 else**:  
 agent.registerInitialState(self.state.deepCopy())  
 *#* ***TODO: bu toplam süreyi aşabilir mi?*** self.unmute()  
  
 agentIndex = self.startingIndex  
 numAgents = len( self.agents )  
  
 **while not** self.gameOver:  
 *# Sonraki ajanı getir* agent = self.agents[agentIndex]  
 move\_time = 0  
 skip\_action = **False  
 if 'observationFunction' in** dir( agent ):  
 self.mute()  
 **if** self.catchExceptions:  
 **try**:  
 timed\_func = TimeoutFunction(agent.observationFunction, int(self.rules.getMoveTimeout(agentIndex)))  
 **try**:  
 start\_time = time.time()  
 observation = timed\_func(self.state.deepCopy())  
 **except** TimeoutFunctionException:  
 skip\_action = **True** move\_time += time.time() - start\_time  
 self.unmute()  
 **except** Exception,data:  
 self.unmute()  
 self.\_agentCrash(agentIndex, quiet=**True**)  
 **return  
 else**:  
 observation = agent.observationFunction(self.state.deepCopy())  
 self.unmute()  
 **else**:  
 observation = self.state.deepCopy()  
  
 *# Bir eylem talep et* action = **None** self.mute()  
 **if** self.catchExceptions:  
 **try**:  
 timed\_func = TimeoutFunction(agent.getAction, int(self.rules.getMoveTimeout(agentIndex)) - int(move\_time))  
 **try**:  
 start\_time = time.time()  
 **if** skip\_action:  
 **raise** TimeoutFunctionException()  
 action = timed\_func( observation )  
 **except** TimeoutFunctionException:  
 print **"Ajan %d tek bir hareketle zaman aşımına uğradı!"** % agentIndex  
 self.agentTimeout = **True** self.unmute()  
 self.\_agentCrash(agentIndex, quiet=**True**)  
 **return** move\_time += time.time() - start\_time  
  
 **if** move\_time > self.rules.getMoveWarningTime(agentIndex):  
 self.totalAgentTimeWarnings[agentIndex] += 1  
 print **"Ajan %d harekete geçmek için çok uzun sürdü! Bu uyarı %d"** % (agentIndex, self.totalAgentTimeWarnings[agentIndex])  
 **if** self.totalAgentTimeWarnings[agentIndex] > self.rules.getMaxTimeWarnings(agentIndex):  
 print **"Ajan %d, maksimum uyarı sayısını aştı: %d"** % (agentIndex, self.totalAgentTimeWarnings[agentIndex])  
 self.agentTimeout = **True** self.unmute()  
 self.\_agentCrash(agentIndex, quiet=**True**)  
  
 self.totalAgentTimes[agentIndex] += move\_time  
 **if** self.totalAgentTimes[agentIndex] > self.rules.getMaxTotalTime(agentIndex):  
 print **"Ajan %d zaman tükendi! (time: %1.2f)"** % (agentIndex, self.totalAgentTimes[agentIndex])  
 self.agentTimeout = **True** self.unmute()  
 self.\_agentCrash(agentIndex, quiet=**True**)  
 **return** self.unmute()  
 **except** Exception,data:  
 self.unmute()  
 self.\_agentCrash(agentIndex)  
 **return  
 else**:  
 action = agent.getAction(observation)  
 self.unmute()  
 self.moveHistory.append( (agentIndex, action) )  
 **if** self.catchExceptions:  
 **try**:  
 self.state = self.state.generateSuccessor( agentIndex, action )  
 **except** Exception,data:  
 self.\_agentCrash(agentIndex)  
 **return  
 else**:  
 self.state = self.state.generateSuccessor( agentIndex, action )  
  
 *# Ekranı değiştir* self.display.update( self.state.data )  
 *# Oyuna özel koşullara izin ver (kazanma, kaybetme vb.)* self.rules.process(self.state, self)  
 *# Parça ilerlemesi* **if** agentIndex == numAgents + 1: self.numMoves += 1  
 *# Sonraki ajan* agentIndex = ( agentIndex + 1 ) % numAgents  
  
 **if** \_BOINC\_ENABLED:  
 boinc.set\_fraction\_done(self.getProgress())  
  
 *# Oyun sonucunu öğrenen bir öğrencisini bilgilendirmek* **for** agent **in** self.agents:  
 **if "final" in** dir( agent ) :  
 **try**:  
 self.mute()  
 agent.final( self.state )  
 self.unmute()  
  
 self.display.finish()

search.py:

**class** SearchProblem:  
 *"""  
 Bu sınıfta hiç bir şeyi değiştirmene gerek yok.  
 """* **def** getStartState(self):  
 *"""  
 Arama sorununun başlangıç ​​durumunu döndürür.  
 """* util.raiseNotDefined()  
  
 **def** isGoalState(self, state):  
 *"""  
 Yalnızca, durum geçerli bir hedef durum olduğunda True değerini döndürür.  
 """* util.raiseNotDefined()  
  
 **def** getSuccessors(self, state):  
 *"""  
 state: Arama durumu  
 """* util.raiseNotDefined()  
  
 **def** getCostOfActions(self, actions):  
 *"""  
 Bu yöntem, belirli bir eylem dizisinin toplam maliyetini döndürür.  
 actions=eylemler anlamında  
 """* util.raiseNotDefined()  
  
  
**def** tinyMazeSearch(problem):  
 *"""  
 TinyMaze'yi çözen bir hareket dizisi döndürür.(küçük labirent)  
 """* **from** game **import** Directions  
 s = Directions.SOUTH  
 w = Directions.WEST  
 **return** [s, s, w, s, w, w, s, w]  
  
  
**def** depthFirstSearch(problem):  
 *"""Önce arama ağacındaki en derin düğümleri arayın."""* loc\_stack = Stack()  
 visited\_node = {}  
 parent\_child\_map = {}  
 direction\_list = []  
  
 start\_node = problem.getStartState()  
 parent\_child\_map[start\_node] = []  
 loc\_stack.push(start\_node)  
  
 **def** traverse\_path(parent\_node):  
 **while True**:  
 map\_row = parent\_child\_map[parent\_node]  
 **if** (len(map\_row) == 2):  
 parent\_node = map\_row[0]  
 direction = map\_row[1]  
 direction\_list.append(direction)  
 **else**:  
 **break  
 return** direction\_list  
  
 **while** (loc\_stack.isEmpty() == **False**):  
  
 parent\_node = loc\_stack.pop()  
  
 **if** (problem.isGoalState(parent\_node)):  
 pathlist = traverse\_path(parent\_node)  
 pathlist.reverse()  
 **return** pathlist  
  
 **elif** (visited\_node.has\_key(parent\_node) == **False**):  
 visited\_node[parent\_node] = []  
 sucessor\_list = problem.getSuccessors(parent\_node)  
 no\_of\_child = len(sucessor\_list)  
 **if** (no\_of\_child > 0):  
 temp = 0  
 **while** (temp < no\_of\_child):  
 child\_nodes = sucessor\_list[temp]  
 child\_state = child\_nodes[0];  
 child\_action = child\_nodes[1];  
 **if** (visited\_node.has\_key(child\_state) == **False**):  
 loc\_stack.push(child\_state)  
 parent\_child\_map[child\_state] = [parent\_node, child\_action]  
 temp = temp + 1  
  
  
**def** breadthFirstSearch(problem):  
 *"""Önce arama ağacındaki en sığ düğümleri arayın."""* loc\_queue = Queue()  
 visited\_node = {}  
 parent\_child\_map = {}  
 direction\_list = []  
  
 start\_node = problem.getStartState()  
 parent\_child\_map[start\_node] = []  
 loc\_queue.push(start\_node)  
  
 **def** traverse\_path(parent\_node):  
 **while True**:  
 map\_row = parent\_child\_map[parent\_node]  
 **if** (len(map\_row) == 2):  
 parent\_node = map\_row[0]  
 direction = map\_row[1]  
 direction\_list.append(direction)  
 **else**:  
 **break  
 return** direction\_list  
  
 **while** (loc\_queue.isEmpty() == **False**):  
  
 parent\_node = loc\_queue.pop()  
  
 **if** (problem.isGoalState(parent\_node)):  
 pathlist = traverse\_path(parent\_node)  
 pathlist.reverse()  
 **return** pathlist  
  
 **elif** (visited\_node.has\_key(parent\_node) == **False**):  
 visited\_node[parent\_node] = []  
 sucessor\_list = problem.getSuccessors(parent\_node)  
 no\_of\_child = len(sucessor\_list)  
 **if** (no\_of\_child > 0):  
 temp = 0  
 **while** (temp < no\_of\_child):  
 child\_nodes = sucessor\_list[temp]  
 child\_state = child\_nodes[0];  
 child\_action = child\_nodes[1];  
 **if** (visited\_node.has\_key(child\_state) == **False**):  
 loc\_queue.push(child\_state)  
 **if** (parent\_child\_map.has\_key(child\_state) == **False**):  
 parent\_child\_map[child\_state] = [parent\_node, child\_action]  
 temp = temp + 1  
  
  
**def** uniformCostSearch(problem):  
 loc\_pqueue = PriorityQueue()  
 visited\_node = {}  
 parent\_child\_map = {}  
 direction\_list = []  
 path\_cost = 0  
  
 start\_node = problem.getStartState()  
 parent\_child\_map[start\_node] = []  
 loc\_pqueue.push(start\_node, path\_cost)  
  
 **def** traverse\_path(parent\_node):  
 **while True**:  
 map\_row = parent\_child\_map[parent\_node]  
 **if** (len(map\_row) == 3):  
 parent\_node = map\_row[0]  
 direction = map\_row[1]  
 direction\_list.append(direction)  
 **else**:  
 **break  
 return** direction\_list  
  
 **while** (loc\_pqueue.isEmpty() == **False**):  
  
 parent\_node = loc\_pqueue.pop()  
  
 **if** (parent\_node != problem.getStartState()):  
 path\_cost = parent\_child\_map[parent\_node][2]  
  
 **if** (problem.isGoalState(parent\_node)):  
 pathlist = traverse\_path(parent\_node)  
 pathlist.reverse()  
 **return** pathlist  
  
 **elif** (visited\_node.has\_key(parent\_node) == **False**):  
 visited\_node[parent\_node] = []  
 sucessor\_list = problem.getSuccessors(parent\_node)  
 no\_of\_child = len(sucessor\_list)  
 **if** (no\_of\_child > 0):  
 temp = 0  
 **while** (temp < no\_of\_child):  
 child\_nodes = sucessor\_list[temp]  
 child\_state = child\_nodes[0];  
 child\_action = child\_nodes[1];  
 child\_cost = child\_nodes[2];  
 gvalue = path\_cost + child\_cost  
 **if** (visited\_node.has\_key(child\_state) == **False**):  
 loc\_pqueue.push(child\_state, gvalue)  
 **if** (parent\_child\_map.has\_key(child\_state) == **False**):  
 parent\_child\_map[child\_state] = [parent\_node, child\_action, gvalue]  
 **else**:  
 **if** (child\_state != start\_node):  
 stored\_cost = parent\_child\_map[child\_state][2]  
 **if** (stored\_cost > gvalue):  
 parent\_child\_map[child\_state] = [parent\_node, child\_action, gvalue]  
 temp = temp + 1  
  
  
**def** nullHeuristic(state, problem=**None**):  
 *"""  
 Sezgisel bir fonksiyon, mevcut durumdan maliyeti en yakın olana kadar tahmin eder.  
 """* **return** 0  
  
  
**def** aStarSearch(problem, heuristic=nullHeuristic):  
 *"""İlk olarak en düşük maliyeti ve sezgisel olan düğümü arayalım."""* loc\_pqueue = PriorityQueue()  
 visited\_node = {}  
 parent\_child\_map = {}  
 direction\_list = []  
 path\_cost = 0  
 heuristic\_value = 0  
  
 start\_node = problem.getStartState()  
 parent\_child\_map[start\_node] = []  
 loc\_pqueue.push(start\_node, heuristic\_value)  
  
 **def** traverse\_path(parent\_node):  
 temp = 0  
 **while True**:  
 **'''print parent\_node'''** map\_row = parent\_child\_map[parent\_node]  
 **if** (len(map\_row) == 4):  
 parent\_node = map\_row[0]  
 direction = map\_row[1]  
 gvalue = map\_row[2]  
 fvalue = map\_row[3]  
 direction\_list.append(direction)  
 **'''print "Gvalue = %d" % gvalue  
 print fvalue'''  
 '''print "Hueristic = %d" % (fvalue-gvalue)'''  
 '''print "Admissible H = %d" % temp'''** temp = temp + 1  
 **else**:  
 **break  
 return** direction\_list  
  
 **while** (loc\_pqueue.isEmpty() == **False**):  
  
 parent\_node = loc\_pqueue.pop()  
  
 **if** (parent\_node != problem.getStartState()):  
 path\_cost = parent\_child\_map[parent\_node][2]  
  
 **if** (problem.isGoalState(parent\_node)):  
 pathlist = traverse\_path(parent\_node)  
 pathlist.reverse()  
 **return** pathlist  
  
 **elif** (visited\_node.has\_key(parent\_node) == **False**):  
 visited\_node[parent\_node] = []  
 sucessor\_list = problem.getSuccessors(parent\_node)  
 no\_of\_child = len(sucessor\_list)  
 **if** (no\_of\_child > 0):  
 temp = 0  
 **while** (temp < no\_of\_child):  
 child\_nodes = sucessor\_list[temp]  
 child\_state = child\_nodes[0];  
 child\_action = child\_nodes[1];  
 child\_cost = child\_nodes[2];  
  
 heuristic\_value = heuristic(child\_state, problem)  
 gvalue = path\_cost + child\_cost  
 fvalue = gvalue + heuristic\_value  
  
 **if** (visited\_node.has\_key(child\_state) == **False**):  
 loc\_pqueue.push(child\_state, fvalue)  
 **if** (parent\_child\_map.has\_key(child\_state) == **False**):  
 parent\_child\_map[child\_state] = [parent\_node, child\_action, gvalue, fvalue]  
 **else**:  
 **if** (child\_state != start\_node):  
 stored\_fvalue = parent\_child\_map[child\_state][3]  
 **if** (stored\_fvalue > fvalue):  
 parent\_child\_map[child\_state] = [parent\_node, child\_action, gvalue, fvalue]  
 temp = temp + 1  
  
  
*#Kısıtlamalar*bfs = breadthFirstSearch  
dfs = depthFirstSearch  
astar = aStarSearch  
ucs = uniformCostSearch

searcAgent.py:

**from** game **import** Directions  
**from** game **import** Agent  
**from** game **import** Actions  
**from** util **import** manhattanDistance  
**from** search **import** breadthFirstSearch  
**import** time  
**import** search  
  
**class** GoWestAgent(Agent):  
  
 **def** getAction(self, state):  
 *"Aracı bir GameState (pacman.py dosyasında tanımlı) alıyor."* **if** Directions.WEST **in** state.getLegalPacmanActions():  
 **return** Directions.WEST  
 **else**:  
 **return** Directions.STOP  
  
**class** SearchAgent(Agent):  
 *"""  
 Varsayılan olarak, bu aracı bulmak için DFS'yi bir PositionSearchProblem üzerinde çalıştırır.  
 Fn için seçenekler şunları içerir:  
 depthFirstSearch veya dfs  
 breadthFirstSearch veya bfs  
 Not: SearchAgent'ta hiçbir kodu DEĞİŞTİRMEMELİDİR  
 """* **def** \_\_init\_\_(self, fn=**'depthFirstSearch'**, prob=**'PositionSearchProblem'**, heuristic=**'nullHeuristic'**):  
  
 *# Uyarı: Doğru fonksiyonları ve problemleri bulmak için bazı gelişmiş Python sihirleri kullanılır.* **if** fn **not in** dir(search):  
 **raise** AttributeError, fn + **'search.pyde bir arama işlevi değil.'** func = getattr(search, fn)  
 **if 'heuristic' not in** func.func\_code.co\_varnames:  
 print(**'[SearchAgent] işlevi kullanmak'** + fn)  
 self.searchFunction = func  
 **else**:  
 **if** heuristic **in** globals().keys():  
 heur = globals()[heuristic]  
 **elif** heuristic **in** dir(search):  
 heur = getattr(search, heuristic)  
 **else**:  
 **raise** AttributeError, heuristic + **' searchAgents.py veya search.pyde bir işlev değil.'** print(**'[SearchAgent] using function %s and heuristic %s'** % (fn, heuristic))  
  
 *# Not: Bu Python hilesi biraz arama algoritması ve sezgisel birleştirir* self.searchFunction = **lambda** x: func(x, heuristic=heur)  
 **if** prob **not in** globals().keys() **or not** prob.endswith(**'Problem'**):  
 **raise** AttributeError, prob + **' SearchAgents.py de bir arama sorunu türü değil.'** self.searchType = globals()[prob]  
 print(**'[SearchAgent] problem türünü kullanma'** + prob)  
  
 **def** registerInitialState(self, state):  
 *"""  
 Burada hedefe giden yolu seçiyoruz. Bu aşamada, ajan  
 Hedefe giden yolu hesaplamalı ve yerel bir değişkende saklamalıdır.  
 """* **if** self.searchFunction == **None**: **raise** Exception, **'SearchAgent için hiçbir arama işlevi sağlanmadı'** starttime = time.time()  
 problem = self.searchType(state) *# Makes a new search problem* self.actions = self.searchFunction(problem) *# Find a path* totalCost = problem.getCostOfActions(self.actions)  
 print(**'Yol toplam maliyeti ile bulundu %d in %.1f seconds'** % (totalCost, time.time() - starttime))  
 **if '\_expanded' in** dir(problem): print(**'Arama düğümleri genişletildi: %d'** % problem.\_expanded)  
  
 **def** getAction(self, state):  
  
 **if 'actionIndex' not in** dir(self): self.actionIndex = 0  
 i = self.actionIndex  
 self.actionIndex += 1  
 **if** i < len(self.actions):  
 **return** self.actions[i]  
 **else**:  
 **return** Directions.STOP  
  
  
**class** PositionSearchProblem(search.SearchProblem):  
  
  
 **def** \_\_init\_\_(self, gameState, costFn=**lambda** x: 1, goal=(1, 1), start=**None**, warn=**True**, visualize=**True**):  
 *"""  
 Başlangıç ​​ve hedefi depolar.  
 gameState: Bir GameState nesnesi  
 costFn: Bir arama durumundan (tuple) negatif olmayan bir sayıya bir işlev  
 goal: gameState'deki bir konum  
 """* self.walls = gameState.getWalls()  
 self.startState = gameState.getPacmanPosition()  
 **if** start != **None**: self.startState = start  
 self.goal = goal  
 self.costFn = costFn  
 self.visualize = visualize  
 **if** warn **and** (gameState.getNumFood() != 1 **or not** gameState.hasFood(\*goal)):  
 print  
 **'Warning: this does not look like a regular search maze'** *# Teşhir amaçlı* self.\_visited, self.\_visitedlist, self.\_expanded = {}, [], 0 *# DO NOT CHANGE* **def** getStartState(self):  
 **return** self.startState  
  
 **def** isGoalState(self, state):  
 isGoal = state == self.goal  
  
 *# Sadece görüntüleme amaçlı* **if** isGoal **and** self.visualize:  
 self.\_visitedlist.append(state)  
 **import** \_\_main\_\_  
 **if '\_display' in** dir(\_\_main\_\_):  
 **if 'drawExpandedCells' in** dir(\_\_main\_\_.\_display): *# @UndefinedVariable* \_\_main\_\_.\_display.drawExpandedCells(self.\_visitedlist) *# @UndefinedVariable* **return** isGoal  
  
 **def** getSuccessors(self, state):  
  
  
 successors = []  
 **for** action **in** [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:  
 x, y = state  
 dx, dy = Actions.directionToVector(action)  
 nextx, nexty = int(x + dx), int(y + dy)  
 **if not** self.walls[nextx][nexty]:  
 nextState = (nextx, nexty)  
 cost = self.costFn(nextState)  
 successors.append((nextState, action, cost))  
  
 *# Görüntüleme amaçlı defter tutma* self.\_expanded += 1 *# DO NOT CHANGE* **if** state **not in** self.\_visited:  
 self.\_visited[state] = **True** self.\_visitedlist.append(state)  
  
 **return** successors  
  
 **def** getCostOfActions(self, actions):  
  
 **if** actions == **None**: **return** 999999  
 x, y = self.getStartState()  
 cost = 0  
 **for** action **in** actions:  
 *# Check figure out the next state and see whether its' legal* dx, dy = Actions.directionToVector(action)  
 x, y = int(x + dx), int(y + dy)  
 **if** self.walls[x][y]: **return** 999999  
 cost += self.costFn((x, y))  
 **return** cost  
  
  
**class** StayEastSearchAgent(SearchAgent):  
  
 **def** \_\_init\_\_(self):  
 self.searchFunction = search.uniformCostSearch  
 costFn = **lambda** pos: .5 \*\* pos[0]  
 self.searchType = **lambda** state: PositionSearchProblem(state, costFn, (1, 1), **None**, **False**)  
  
  
**class** StayWestSearchAgent(SearchAgent):  
  
  
 **def** \_\_init\_\_(self):  
 self.searchFunction = search.uniformCostSearch  
 costFn = **lambda** pos: 2 \*\* pos[0]  
 self.searchType = **lambda** state: PositionSearchProblem(state, costFn)  
  
  
**def** manhattanHeuristic(position, problem, info={}):  
 *"""manhattan uzaklığı"""* xy1 = position  
 xy2 = problem.goal  
 **return** abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])  
  
  
**def** euclideanHeuristic(position, problem, info={}):  
  
 xy1 = position  
 xy2 = problem.goal  
 **return** ((xy1[0] - xy2[0]) \*\* 2 + (xy1[1] - xy2[1]) \*\* 2) \*\* 0.5  
  
**class** CornersProblem(search.SearchProblem):  
  
 **def** \_\_init\_\_(self, startingGameState):  
  
 self.walls = startingGameState.getWalls()  
 self.startingPosition = startingGameState.getPacmanPosition()  
 top, right = self.walls.height - 2, self.walls.width - 2  
 self.corners = ((1, 1), (1, top), (right, 1), (right, top))  
 **for** corner **in** self.corners:  
 **if not** startingGameState.hasFood(\*corner):  
 print  
 **'Uyarı: köşede yiyecek yok '** + str(corner)  
 self.\_expanded = 0 *# DEĞİŞTİRME; Genişletilmiş genişletilmiş düğüm sayısı* **"\*\*\* Kod burada \*\*\*"** self.Corner1 = **False** self.Corner2 = **False** self.Corner3 = **False** self.Corner4 = **False  
 '''self.corner\_visited = 0'''  
  
 def** getStartState(self):  
 **return** self.startingPosition  
  
 **def** isGoalState(self, state):  
 *"""Bu arama durumunun sorunun bir amaç durumu olup olmadığını döndürür. """* **if not** (state == self.startingPosition):  
 corner\_visited = state[1]  
 **if** (corner\_visited == (**True**, **True**, **True**, **True**)):  
 **return** state  
  
 **def** getSuccessors(self, state):  
  
 successors = []  
  
 **for** action **in** [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:  
  
 **if not** (state == self.startingPosition):  
 postition\_state = state[0]  
 self.Corner1 = state[1][0]  
 self.Corner2 = state[1][1]  
 self.Corner3 = state[1][2]  
 self.Corner4 = state[1][3]  
 **else**:  
 postition\_state = state  
 self.Corner1 = **False** self.Corner2 = **False** self.Corner3 = **False** self.Corner4 = **False** x, y = postition\_state  
 dx, dy = Actions.directionToVector(action)  
 nextx, nexty = int(x + dx), int(y + dy)  
 **if not** self.walls[nextx][nexty]:  
 cost = 1  
 new\_position = (nextx, nexty)  
 **if** (new\_position == self.corners[0]):  
 self.Corner1 = **True  
 if** (new\_position == self.corners[1]):  
 self.Corner2 = **True  
 if** (new\_position == self.corners[2]):  
 self.Corner3 = **True  
 if** (new\_position == self.corners[3]):  
 self.Corner4 = **True** nextState = (new\_position, (self.Corner1, self.Corner2, self.Corner3, self.Corner4))  
 successors.append((nextState, action, cost))  
  
 self.\_expanded += 1  
 **return** successors  
  
 **def** getCostOfActions(self, actions):  
 *"""maliyetini döndürür"""* **if** actions == **None**: **return** 999999  
 x, y = self.startingPosition  
 **for** action **in** actions:  
 dx, dy = Actions.directionToVector(action)  
 x, y = int(x + dx), int(y + dy)  
 **if** self.walls[x][y]: **return** 999999  
 **return** len(actions)  
  
  
**def** cornersHeuristic(state, problem):  
 *"""  
 Tanımladığınız CornersProblem için bir sezgisel.  
 Bu işlev her zaman alt sınırı olan bir sayıyı döndürmelidir. sorunun en kısa yolu olmalı(aynı zamanda tutarlı).  
 """* corners = problem.corners *# Bunlar köşe koordinatları* walls = problem.walls *# labirent duvarlarıdır* corner\_unvisited = []  
 total\_distance = 0  
  
 **if not** (state == problem.startingPosition):  
 postition\_state = state[0]  
 **if** (state[1][0] == **False**):  
 corner\_unvisited.append(problem.corners[0])  
 **if** (state[1][1] == **False**):  
 corner\_unvisited.append(problem.corners[1])  
 **if** (state[1][2] == **False**):  
 corner\_unvisited.append(problem.corners[2])  
 **if** (state[1][3] == **False**):  
 corner\_unvisited.append(problem.corners[3])  
 **else**:  
 postition\_state = state  
 corner\_unvisited = list(problem.corners)  
  
 **if** (len(corner\_unvisited) == 0):  
 **return** 0  
  
 **while** (len(corner\_unvisited) > 0):  
 distance\_to\_all\_corners = []  
 **for** selected\_corner **in** corner\_unvisited:  
 xy1 = postition\_state  
 xy2 = selected\_corner  
 distance\_to\_corner = abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])  
 distance\_to\_all\_corners.append((distance\_to\_corner, selected\_corner))  
  
 closest\_corner = min(distance\_to\_all\_corners)  
 corner\_unvisited.remove(closest\_corner[1])  
 postition\_state = closest\_corner[1]  
 total\_distance = total\_distance + closest\_corner[0]  
  
 **return** total\_distance *# Default to trivial solution***class** AStarCornersAgent(SearchAgent):  
  
  
 **def** \_\_init\_\_(self):  
 self.searchFunction = **lambda** prob: search.aStarSearch(prob, cornersHeuristic)  
 self.searchType = CornersProblem  
  
  
**class** FoodSearchProblem:  
 *"""  
 foodGrid: Kalan yiyeceği belirten Doğru veya Yanlış olan bir Kılavuz  
 """* **def** \_\_init\_\_(self, startingGameState):  
 self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())  
 self.walls = startingGameState.getWalls()  
 self.startingGameState = startingGameState  
 self.\_expanded = 0 *# DO NOT CHANGE* self.heuristicInfo = {} *# A dictionary for the heuristic to store information* **def** getStartState(self):  
 **return** self.start  
  
 **def** isGoalState(self, state):  
 **return** state[1].count() == 0  
  
 **def** getSuccessors(self, state):  
 successors = []  
 self.\_expanded += 1 *# DO NOT CHANGE* **for** direction **in** [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:  
 x, y = state[0]  
 dx, dy = Actions.directionToVector(direction)  
 nextx, nexty = int(x + dx), int(y + dy)  
 **if not** self.walls[nextx][nexty]:  
 nextFood = state[1].copy()  
 nextFood[nextx][nexty] = **False** successors.append((((nextx, nexty), nextFood), direction, 1))  
 **return** successors  
  
 **def** getCostOfActions(self, actions):  
 *""" Belirli bir eylem dizisinin maliyetini döndürür."""* x, y = self.getStartState()[0]  
 cost = 0  
 **for** action **in** actions:  
 *# figure out the next state and see whether it's legal* dx, dy = Actions.directionToVector(action)  
 x, y = int(x + dx), int(y + dy)  
 **if** self.walls[x][y]:  
 **return** 999999  
 cost += 1  
 **return** cost  
  
  
**class** AStarFoodSearchAgent(SearchAgent):  
  
  
 **def** \_\_init\_\_(self):  
 self.searchFunction = **lambda** prob: search.aStarSearch(prob, foodHeuristic)  
 self.searchType = FoodSearchProblem  
  
  
**def** foodHeuristic(state, problem):  
 *"""  
 A \* kullanılıyorsa, daha kötü bir maliyet araştırması buluntusu olan bir çözüm bulur.  
 """* position, foodGrid = state  
 hvalue = 0  
 food\_available = []  
 total\_distance = 0  
 **for** i **in** range(0, foodGrid.width):  
 **for** j **in** range(0, foodGrid.height):  
 **if** (foodGrid[i][j] == **True**):  
 food\_location = (i, j)  
 food\_available.append(food\_location)  
  
 **if** (len(food\_available) == 0):  
 **return** 0  
  
 max\_distance = ((0, 0), (0, 0), 0)  
  
 **for** current\_food **in** food\_available:  
 **for** select\_food **in** food\_available:  
 **if** (current\_food == select\_food):  
 **pass  
 else**:  
 distance = manhattanDistance(current\_food, select\_food)  
 **if** (max\_distance[2] < distance):  
 max\_distance = (current\_food, select\_food, distance)  
  
 **if** (max\_distance[0] == (0, 0) **and** max\_distance[1] == (0, 0)):  
 hvalue = manhattanDistance(position, food\_available[0])  
 **else**:  
 d1 = manhattanDistance(position, max\_distance[0])  
 d2 = manhattanDistance(position, max\_distance[1])  
 hvalue = max\_distance[2] + min(d1, d2)  
  
 **return** hvalue  
  
  
**class** ClosestDotSearchAgent(SearchAgent):  
 *"""Bir dizi arama kullanarak tüm yiyecekleri ara"""* **def** registerInitialState(self, state):  
 self.actions = []  
 currentState = state  
 **while** (currentState.getFood().count() > 0):  
 nextPathSegment = self.findPathToClosestDot(currentState) *# The missing piece* self.actions += nextPathSegment  
 **for** action **in** nextPathSegment:  
 legal = currentState.getLegalActions()  
 **if** action **not in** legal:  
 t = (str(action), str(currentState))  
 **raise** Exception, **'findPathToClosestDot returned an illegal move: %s!\n%s'** % t  
 currentState = currentState.generateSuccessor(0, action)  
 self.actionIndex = 0  
 print  
 **'Path found with cost %d.'** % len(self.actions)  
  
 **def** findPathToClosestDot(self, gameState):  
 *"""  
 En yakın noktaya giden bir yolu döndürür.  
 """* startPosition = gameState.getPacmanPosition()  
 food = gameState.getFood()  
 walls = gameState.getWalls()  
 problem = AnyFoodSearchProblem(gameState)  
  
 action\_list = breadthFirstSearch(problem)  
  
 **return** action\_list  
  
  
**class** AnyFoodSearchProblem(PositionSearchProblem):  
 *"""Herhangi bir yemek için bir yol bulmak için bir arama problemi."""* **def** \_\_init\_\_(self, gameState):  
 *"Bilgileri gameState'den depolar. Bunu değiştirmenize gerek yoktur."  
 # Daha sonra başvurmak için yiyecekleri saklayın* self.food = gameState.getFood()  
 self.walls = gameState.getWalls()  
 self.startState = gameState.getPacmanPosition()  
 self.costFn = **lambda** x: 1  
 self.\_visited, self.\_visitedlist, self.\_expanded = {}, [], 0 *# DO NOT CHANGE* **def** isGoalState(self, state):  
  
 x, y = state  
 foodGrid = self.food  
 **if** (foodGrid[x][y] == **True**) **or** (foodGrid.count() == 0):  
 **return True  
  
  
def** mazeDistance(point1, point2, gameState):  
 *"""  
 Arama fonksiyonlarını kullanarak herhangi iki nokta arasındaki labirent mesafesini döndürür.  
 """* x1, y1 = point1  
 x2, y2 = point2  
 walls = gameState.getWalls()  
 **assert not** walls[x1][y1], **'point1 is a wall: '** + str(point1)  
 **assert not** walls[x2][y2], **'point2 is a wall: '** + str(point2)  
 prob = PositionSearchProblem(gameState, start=point1, goal=point2, warn=**False**, visualize=**False**)  
 **return** len(search.bfs(prob))

KeyboardAgent.py:

**from** game **import** Agent  
**from** game **import** Directions  
**import** random  
  
  
**class** KeyboardAgent(Agent):  
 WEST\_KEY = **'a'** EAST\_KEY = **'d'** NORTH\_KEY = **'w'** SOUTH\_KEY = **'s'** STOP\_KEY = **'q'  
  
 def** \_\_init\_\_(self, index=0):  
  
 self.lastMove = Directions.STOP  
 self.index = index  
 self.keys = []  
  
 **def** getAction(self, state):  
 **from** graphicsUtils **import** keys\_waiting  
 **from** graphicsUtils **import** keys\_pressed  
 keys = keys\_waiting() + keys\_pressed()  
 **if** keys != []:  
 self.keys = keys  
  
 legal = state.getLegalActions(self.index)  
 move = self.getMove(legal)  
  
 **if** move == Directions.STOP:  
 **if** self.lastMove **in** legal:  
 move = self.lastMove  
  
 **if** (self.STOP\_KEY **in** self.keys) **and** Directions.STOP **in** legal:  
 move = Directions.STOP  
  
 **if** move **not in** legal:  
 move = random.choice(legal)  
  
 self.lastMove = move  
 **return** move  
  
 **def** getMove(self, legal):  
 move = Directions.STOP  
 **if** (self.WEST\_KEY **in** self.keys **or 'Left' in** self.keys) **and** Directions.WEST **in** legal:  
 move = Directions.WEST  
 **if** (self.EAST\_KEY **in** self.keys **or 'Right' in** self.keys) **and** Directions.EAST **in** legal:  
 move = Directions.EAST  
 **if** (self.NORTH\_KEY **in** self.keys **or 'Up' in** self.keys) **and** Directions.NORTH **in** legal:  
 move = Directions.NORTH  
 **if** (self.SOUTH\_KEY **in** self.keys **or 'Down' in** self.keys) **and** Directions.SOUTH **in** legal:  
 move = Directions.SOUTH  
 **return** move  
  
  
**class** KeyboardAgent2(KeyboardAgent):  
  
 WEST\_KEY = **'j'** EAST\_KEY = **"l"** NORTH\_KEY = **'i'** SOUTH\_KEY = **'k'** STOP\_KEY = **'u'  
  
 def** getMove(self, legal):  
 move = Directions.STOP  
 **if** (self.WEST\_KEY **in** self.keys) **and** Directions.WEST **in** legal:  
 move = Directions.WEST  
 **if** (self.EAST\_KEY **in** self.keys) **and** Directions.EAST **in** legal:  
 move = Directions.EAST  
 **if** (self.NORTH\_KEY **in** self.keys) **and** Directions.NORTH **in** legal:  
 move = Directions.NORTH  
 **if** (self.SOUTH\_KEY **in** self.keys) **and** Directions.SOUTH **in** legal:  
 move = Directions.SOUTH  
 **return** move

graphicsDisplay.py;

**import** math  
**import** time  
  
  
  
DEFAULT\_GRID\_SIZE = 30.0  
INFO\_PANE\_HEIGHT = 35  
BACKGROUND\_COLOR = formatColor(0, 0, 0)  
WALL\_COLOR = formatColor(0.0 / 255.0, 51.0 / 255.0, 255.0 / 255.0)  
INFO\_PANE\_COLOR = formatColor(.4, .4, 0)  
SCORE\_COLOR = formatColor(.9, .9, .9)  
PACMAN\_OUTLINE\_WIDTH = 2  
PACMAN\_CAPTURE\_OUTLINE\_WIDTH = 4  
  
GHOST\_COLORS = []  
GHOST\_COLORS.append(formatColor(.9, 0, 0)) *# Red*GHOST\_COLORS.append(formatColor(0, .3, .9)) *# Blue*GHOST\_COLORS.append(formatColor(.98, .41, .07)) *# Orange*GHOST\_COLORS.append(formatColor(.1, .75, .7)) *# Green*GHOST\_COLORS.append(formatColor(1.0, 0.6, 0.0)) *# Yellow*GHOST\_COLORS.append(formatColor(.4, 0.13, 0.91)) *# Purple*TEAM\_COLORS = GHOST\_COLORS[:2]  
  
GHOST\_SHAPE = [  
 (0, 0.3),  
 (0.25, 0.75),  
 (0.5, 0.3),  
 (0.75, 0.75),  
 (0.75, -0.5),  
 (0.5, -0.75),  
 (-0.5, -0.75),  
 (-0.75, -0.5),  
 (-0.75, 0.75),  
 (-0.5, 0.3),  
 (-0.25, 0.75)  
]  
GHOST\_SIZE = 0.65  
SCARED\_COLOR = formatColor(1, 1, 1)  
  
GHOST\_VEC\_COLORS = list(map(colorToVector, GHOST\_COLORS))  
  
PACMAN\_COLOR = formatColor(255.0 / 255.0, 255.0 / 255.0, 61.0 / 255)  
PACMAN\_SCALE = 0.5  
  
FOOD\_COLOR = formatColor(1, 1, 1)  
FOOD\_SIZE = 0.1  
  
LASER\_COLOR = formatColor(1, 0, 0)  
LASER\_SIZE = 0.02  
  
CAPSULE\_COLOR = formatColor(1, 1, 1)  
CAPSULE\_SIZE = 0.25  
  
WALL\_RADIUS = 0.15  
  
  
**class** InfoPane:  
  
 **def** \_\_init\_\_(self, layout, gridSize):  
 self.gridSize = gridSize  
 self.width = (layout.width) \* gridSize  
 self.base = (layout.height + 1) \* gridSize  
 self.height = INFO\_PANE\_HEIGHT  
 self.fontSize = 24  
 self.textColor = PACMAN\_COLOR  
 self.drawPane()  
  
 **def** toScreen(self, pos, y=**None**):  
 **if** y == **None**:  
 x, y = pos  
 **else**:  
 x = pos  
  
 x = self.gridSize + x  
 y = self.base + y  
 **return** x, y  
  
 **def** drawPane(self):  
 self.scoreText = text(self.toScreen(  
 0, 0), self.textColor, **"SCORE: 0"**, **"Times"**, self.fontSize, **"bold"**)  
  
 **def** initializeGhostDistances(self, distances):  
 self.ghostDistanceText = []  
  
 size = 20  
 **if** self.width < 240:  
 size = 12  
 **if** self.width < 160:  
 size = 10  
  
 **for** i, d **in** enumerate(distances):  
 t = text(self.toScreen(self.width / 2 + self.width / 8 \* i,  
 0), GHOST\_COLORS[i + 1], d, **"Times"**, size, **"bold"**)  
 self.ghostDistanceText.append(t)  
  
 **def** updateScore(self, score):  
 changeText(self.scoreText, **"SCORE: % 4d"** % score)  
  
 **def** setTeam(self, isBlue):  
 text = **"RED TEAM"  
 if** isBlue:  
 text = **"BLUE TEAM"** self.teamText = text(self.toScreen(  
 300, 0), self.textColor, text, **"Times"**, self.fontSize, **"bold"**)  
  
 **def** updateGhostDistances(self, distances):  
 **if** len(distances) == 0:  
 **return  
 if 'ghostDistanceText' not in** dir(self):  
 self.initializeGhostDistances(distances)  
 **else**:  
 **for** i, d **in** enumerate(distances):  
 changeText(self.ghostDistanceText[i], d)  
  
 **def** drawGhost(self):  
 **pass  
  
 def** drawPacman(self):  
 **pass  
  
 def** drawWarning(self):  
 **pass  
  
 def** clearIcon(self):  
 **pass  
  
 def** updateMessage(self, message):  
 **pass  
  
 def** clearMessage(self):  
 **pass  
  
  
class** PacmanGraphics:  
  
 **def** \_\_init\_\_(self, zoom=1.0, frameTime=0.0, capture=**False**):  
 self.have\_window = 0  
 self.currentGhostImages = {}  
 self.pacmanImage = **None** self.zoom = zoom  
 self.gridSize = DEFAULT\_GRID\_SIZE \* zoom  
 self.capture = capture  
 self.frameTime = frameTime  
  
 **def** checkNullDisplay(self):  
 **return False  
  
 def** initialize(self, state, isBlue=**False**):  
 self.isBlue = isBlue  
 self.startGraphics(state)  
 self.distributionImages = **None** self.drawStaticObjects(state)  
 self.drawAgentObjects(state)  
  
 *# Information* self.previousState = state  
  
 **def** startGraphics(self, state):  
 self.layout = state.layout  
 layout = self.layout  
 self.width = layout.width  
 self.height = layout.height  
 self.make\_window(self.width, self.height)  
 self.infoPane = InfoPane(layout, self.gridSize)  
 self.currentState = layout  
  
 **def** drawDistributions(self, state):  
 walls = state.layout.walls  
 dist = []  
 **for** x **in** range(walls.width):  
 distx = []  
 dist.append(distx)  
 **for** y **in** range(walls.height):  
 (screen\_x, screen\_y) = self.to\_screen((x, y))  
 block = square((screen\_x, screen\_y),  
 0.5 \* self.gridSize,  
 color=BACKGROUND\_COLOR,  
 filled=1, behind=2)  
 distx.append(block)  
 self.distributionImages = dist  
  
 **def** drawStaticObjects(self, state):  
 layout = self.layout  
 self.drawWalls(layout.walls)  
 self.food = self.drawFood(layout.food)  
 self.capsules = self.drawCapsules(layout.capsules)  
 refresh()  
  
 **def** drawAgentObjects(self, state):  
 self.agentImages = [] *# (agentState, image)* **for** index, agent **in** enumerate(state.agentStates):  
 **if** agent.isPacman:  
 image = self.drawPacman(agent, index)  
 self.agentImages.append((agent, image))  
 **else**:  
 image = self.drawGhost(agent, index)  
 self.agentImages.append((agent, image))  
 refresh()  
  
 **def** swapImages(self, agentIndex, newState):  
 prevState, prevImage = self.agentImages[agentIndex]  
 **for** item **in** prevImage:  
 remove\_from\_screen(item)  
 **if** newState.isPacman:  
 image = self.drawPacman(newState, agentIndex)  
 self.agentImages[agentIndex] = (newState, image)  
 **else**:  
 image = self.drawGhost(newState, agentIndex)  
 self.agentImages[agentIndex] = (newState, image)  
 refresh()  
  
 **def** update(self, newState):  
 agentIndex = newState.\_agentMoved  
 agentState = newState.agentStates[agentIndex]  
  
 **if** self.agentImages[agentIndex][0].isPacman != agentState.isPacman:  
 self.swapImages(agentIndex, agentState)  
 prevState, prevImage = self.agentImages[agentIndex]  
 **if** agentState.isPacman:  
 self.animatePacman(agentState, prevState, prevImage)  
 **else**:  
 self.moveGhost(agentState, agentIndex, prevState, prevImage)  
 self.agentImages[agentIndex] = (agentState, prevImage)  
  
 **if** newState.\_foodEaten != **None**:  
 self.removeFood(newState.\_foodEaten, self.food)  
 **if** newState.\_capsuleEaten != **None**:  
 self.removeCapsule(newState.\_capsuleEaten, self.capsules)  
 self.infoPane.updateScore(newState.score)  
 **if 'ghostDistances' in** dir(newState):  
 self.infoPane.updateGhostDistances(newState.ghostDistances)  
  
 **def** make\_window(self, width, height):  
 grid\_width = (width - 1) \* self.gridSize  
 grid\_height = (height - 1) \* self.gridSize  
 screen\_width = 2 \* self.gridSize + grid\_width  
 screen\_height = 2 \* self.gridSize + grid\_height + INFO\_PANE\_HEIGHT  
  
 begin\_graphics(screen\_width,  
 screen\_height,  
 BACKGROUND\_COLOR,  
 **"CS188 Pacman"**)  
  
 **def** drawPacman(self, pacman, index):  
 position = self.getPosition(pacman)  
 screen\_point = self.to\_screen(position)  
 endpoints = self.getEndpoints(self.getDirection(pacman))  
  
 width = PACMAN\_OUTLINE\_WIDTH  
 outlineColor = PACMAN\_COLOR  
 fillColor = PACMAN\_COLOR  
  
 **if** self.capture:  
 outlineColor = TEAM\_COLORS[index % 2]  
 fillColor = GHOST\_COLORS[index]  
 width = PACMAN\_CAPTURE\_OUTLINE\_WIDTH  
  
 **return** [circle(screen\_point, PACMAN\_SCALE \* self.gridSize,  
 fillColor=fillColor, outlineColor=outlineColor,  
 endpoints=endpoints,  
 width=width)]  
  
 **def** getEndpoints(self, direction, position=(0, 0)):  
 x, y = position  
 pos = x - int(x) + y - int(y)  
 width = 30 + 80 \* math.sin(math.pi \* pos)  
  
 delta = width / 2  
 **if** (direction == **'West'**):  
 endpoints = (180 + delta, 180 - delta)  
 **elif** (direction == **'North'**):  
 endpoints = (90 + delta, 90 - delta)  
 **elif** (direction == **'South'**):  
 endpoints = (270 + delta, 270 - delta)  
 **else**:  
 endpoints = (0 + delta, 0 - delta)  
 **return** endpoints  
  
 **def** movePacman(self, position, direction, image):  
 screenPosition = self.to\_screen(position)  
 endpoints = self.getEndpoints(direction, position)  
 r = PACMAN\_SCALE \* self.gridSize  
 moveCircle(image[0], screenPosition, r, endpoints)  
 refresh()  
  
 **def** animatePacman(self, pacman, prevPacman, image):  
 **if** self.frameTime < 0:  
 print(**'Press any key to step forward, "q" to play'**)  
 keys = wait\_for\_keys()  
 **if 'q' in** keys:  
 self.frameTime = 0.1  
 **if** self.frameTime > 0.01 **or** self.frameTime < 0:  
 start = time.time()  
 fx, fy = self.getPosition(prevPacman)  
 px, py = self.getPosition(pacman)  
 frames = 4.0  
 **for** i **in** range(1, int(frames) + 1):  
 pos = px \* i / frames + fx \* \  
 (frames - i) / frames, py \* i / \  
 frames + fy \* (frames - i) / frames  
 self.movePacman(pos, self.getDirection(pacman), image)  
 refresh()  
 sleep(abs(self.frameTime) / frames)  
 **else**:  
 self.movePacman(self.getPosition(pacman),  
 self.getDirection(pacman), image)  
 refresh()  
  
 **def** getGhostColor(self, ghost, ghostIndex):  
 **if** ghost.scaredTimer > 0:  
 **return** SCARED\_COLOR  
 **else**:  
 **return** GHOST\_COLORS[ghostIndex]  
  
 **def** drawGhost(self, ghost, agentIndex):  
 pos = self.getPosition(ghost)  
 dir = self.getDirection(ghost)  
 (screen\_x, screen\_y) = (self.to\_screen(pos))  
 coords = []  
 **for** (x, y) **in** GHOST\_SHAPE:  
 coords.append((x \* self.gridSize \* GHOST\_SIZE + screen\_x,  
 y \* self.gridSize \* GHOST\_SIZE + screen\_y))  
  
 colour = self.getGhostColor(ghost, agentIndex)  
 body = polygon(coords, colour, filled=1)  
 WHITE = formatColor(1.0, 1.0, 1.0)  
 BLACK = formatColor(0.0, 0.0, 0.0)  
  
 dx = 0  
 dy = 0  
 **if** dir == **'North'**:  
 dy = -0.2  
 **if** dir == **'South'**:  
 dy = 0.2  
 **if** dir == **'East'**:  
 dx = 0.2  
 **if** dir == **'West'**:  
 dx = -0.2  
 leftEye = circle((screen\_x + self.gridSize \* GHOST\_SIZE \* (-0.3 + dx / 1.5), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy / 1.5)), self.gridSize \* GHOST\_SIZE \* 0.2, WHITE, WHITE)  
 rightEye = circle((screen\_x + self.gridSize \* GHOST\_SIZE \* (0.3 + dx / 1.5), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy / 1.5)), self.gridSize \* GHOST\_SIZE \* 0.2, WHITE, WHITE)  
 leftPupil = circle((screen\_x + self.gridSize \* GHOST\_SIZE \* (-0.3 + dx), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy)), self.gridSize \* GHOST\_SIZE \* 0.08, BLACK, BLACK)  
 rightPupil = circle((screen\_x + self.gridSize \* GHOST\_SIZE \* (0.3 + dx), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy)), self.gridSize \* GHOST\_SIZE \* 0.08, BLACK, BLACK)  
 ghostImageParts = []  
 ghostImageParts.append(body)  
 ghostImageParts.append(leftEye)  
 ghostImageParts.append(rightEye)  
 ghostImageParts.append(leftPupil)  
 ghostImageParts.append(rightPupil)  
  
 **return** ghostImageParts  
  
 **def** moveEyes(self, pos, dir, eyes):  
 (screen\_x, screen\_y) = (self.to\_screen(pos))  
 dx = 0  
 dy = 0  
 **if** dir == **'North'**:  
 dy = -0.2  
 **if** dir == **'South'**:  
 dy = 0.2  
 **if** dir == **'East'**:  
 dx = 0.2  
 **if** dir == **'West'**:  
 dx = -0.2  
 moveCircle(eyes[0], (screen\_x + self.gridSize \* GHOST\_SIZE \* (-0.3 + dx / 1.5), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy / 1.5)), self.gridSize \* GHOST\_SIZE \* 0.2)  
 moveCircle(eyes[1], (screen\_x + self.gridSize \* GHOST\_SIZE \* (0.3 + dx / 1.5), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy / 1.5)), self.gridSize \* GHOST\_SIZE \* 0.2)  
 moveCircle(eyes[2], (screen\_x + self.gridSize \* GHOST\_SIZE \* (-0.3 + dx), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy)), self.gridSize \* GHOST\_SIZE \* 0.08)  
 moveCircle(eyes[3], (screen\_x + self.gridSize \* GHOST\_SIZE \* (0.3 + dx), screen\_y -  
 self.gridSize \* GHOST\_SIZE \* (0.3 - dy)), self.gridSize \* GHOST\_SIZE \* 0.08)  
  
 **def** moveGhost(self, ghost, ghostIndex, prevGhost, ghostImageParts):  
 old\_x, old\_y = self.to\_screen(self.getPosition(prevGhost))  
 new\_x, new\_y = self.to\_screen(self.getPosition(ghost))  
 delta = new\_x - old\_x, new\_y - old\_y  
  
 **for** ghostImagePart **in** ghostImageParts:  
 move\_by(ghostImagePart, delta)  
 refresh()  
  
 **if** ghost.scaredTimer > 0:  
 color = SCARED\_COLOR  
 **else**:  
 color = GHOST\_COLORS[ghostIndex]  
 edit(ghostImageParts[0], (**'fill'**, color), (**'outline'**, color))  
 self.moveEyes(self.getPosition(ghost),  
 self.getDirection(ghost), ghostImageParts[-4:])  
 refresh()  
  
 **def** getPosition(self, agentState):  
 **if** agentState.configuration == **None**:  
 **return** (-1000, -1000)  
 **return** agentState.getPosition()  
  
 **def** getDirection(self, agentState):  
 **if** agentState.configuration == **None**:  
 **return** Directions.STOP  
 **return** agentState.configuration.getDirection()  
  
 **def** finish(self):  
 end\_graphics()  
  
 **def** to\_screen(self, point):  
 (x, y) = point  
 x = (x + 1) \* self.gridSize  
 y = (self.height - y) \* self.gridSize  
 **return** (x, y)  
 **def** to\_screen2(self, point):  
 (x, y) = point  
 *#y = self.height - y* x = (x + 1) \* self.gridSize  
 y = (self.height - y) \* self.gridSize  
 **return** (x, y)  
  
 **def** isWall(self, x, y, walls):  
 **if** x < 0 **or** y < 0:  
 **return False  
 if** x >= walls.width **or** y >= walls.height:  
 **return False  
 return** walls[x][y]  
  
 **def** drawFood(self, foodMatrix):  
 foodImages = []  
 color = FOOD\_COLOR  
 **for** xNum, x **in** enumerate(foodMatrix):  
 **if** self.capture **and** (xNum \* 2) <= foodMatrix.width:  
 color = TEAM\_COLORS[0]  
 **if** self.capture **and** (xNum \* 2) > foodMatrix.width:  
 color = TEAM\_COLORS[1]  
 imageRow = []  
 foodImages.append(imageRow)  
 **for** yNum, cell **in** enumerate(x):  
 **if** cell:  
 screen = self.to\_screen((xNum, yNum))  
 dot = circle(screen,  
 FOOD\_SIZE \* self.gridSize,  
 outlineColor=color, fillColor=color,  
 width=1)  
 imageRow.append(dot)  
 **else**:  
 imageRow.append(**None**)  
 **return** foodImages  
  
 **def** drawCapsules(self, capsules):  
 capsuleImages = {}  
 **for** capsule **in** capsules:  
 (screen\_x, screen\_y) = self.to\_screen(capsule)  
 dot = circle((screen\_x, screen\_y),  
 CAPSULE\_SIZE \* self.gridSize,  
 outlineColor=CAPSULE\_COLOR,  
 fillColor=CAPSULE\_COLOR,  
 width=1)  
 capsuleImages[capsule] = dot  
 **return** capsuleImages  
  
 **def** removeFood(self, cell, foodImages):  
 x, y = cell  
 remove\_from\_screen(foodImages[x][y])  
  
 **def** removeCapsule(self, cell, capsuleImages):  
 x, y = cell  
 remove\_from\_screen(capsuleImages[(x, y)])  
  
 **def** drawExpandedCells(self, cells):  
 n = float(len(cells))  
 baseColor = [1.0, 0.0, 0.0]  
 self.clearExpandedCells()  
 self.expandedCells = []  
 **for** k, cell **in** enumerate(cells):  
 screenPos = self.to\_screen(cell)  
 cellColor = formatColor(  
 \*[(n - k) \* c \* .5 / n + .25 **for** c **in** baseColor])  
 block = square(screenPos,  
 0.5 \* self.gridSize,  
 color=cellColor,  
 filled=1, behind=2)  
 self.expandedCells.append(block)  
 **if** self.frameTime < 0:  
 refresh()  
  
 **def** clearExpandedCells(self):  
 **if 'expandedCells' in** dir(self) **and** len(self.expandedCells) > 0:  
 **for** cell **in** self.expandedCells:  
 remove\_from\_screen(cell)  
  
 **def** updateDistributions(self, distributions):  
 distributions = [x.copy() **for** x **in** distributions]  
 **if** self.distributionImages == **None**:  
 self.drawDistributions(self.previousState)  
 **for** x **in** range(len(self.distributionImages)):  
 **for** y **in** range(len(self.distributionImages[0])):  
 image = self.distributionImages[x][y]  
 weights = [dist[(x, y)] **for** dist **in** distributions]  
  
 **if** sum(weights) != 0:  
 **pass** color = [0.0, 0.0, 0.0]  
 colors = GHOST\_VEC\_COLORS[1:]  
 **if** self.capture:  
 colors = GHOST\_VEC\_COLORS  
 **for** weight, gcolor **in** zip(weights, colors):  
 color = [min(1.0, c + 0.95 \* g \* weight \*\* .3)  
 **for** c, g **in** zip(color, gcolor)]  
 changeColor(image, formatColor(\*color))  
 refresh()  
  
  
**class** FirstPersonPacmanGraphics(PacmanGraphics):  
  
 **def** \_\_init\_\_(self, zoom=1.0, showGhosts=**True**, capture=**False**, frameTime=0):  
 PacmanGraphics.\_\_init\_\_(self, zoom, frameTime=frameTime)  
 self.showGhosts = showGhosts  
 self.capture = capture  
  
 **def** initialize(self, state, isBlue=**False**):  
  
 self.isBlue = isBlue  
 PacmanGraphics.startGraphics(self, state)  
 walls = state.layout.walls  
 dist = []  
 self.layout = state.layout  
 self.distributionImages = **None** self.drawStaticObjects(state)  
 self.drawAgentObjects(state)  
 self.previousState = state  
  
 **def** lookAhead(self, config, state):  
 **if** config.getDirection() == **'Stop'**:  
 **return  
 else**:  
 **pass** allGhosts = state.getGhostStates()  
 visibleGhosts = state.getVisibleGhosts()  
 **for** i, ghost **in** enumerate(allGhosts):  
 **if** ghost **in** visibleGhosts:  
 self.drawGhost(ghost, i)  
 **else**:  
 self.currentGhostImages[i] = **None  
  
 def** getGhostColor(self, ghost, ghostIndex):  
 **return** GHOST\_COLORS[ghostIndex]  
  
 **def** getPosition(self, ghostState):  
 **if not** self.showGhosts **and not** ghostState.isPacman **and** ghostState.getPosition()[1] > 1:  
 **return** (-1000, -1000)  
 **else**:  
 **return** PacmanGraphics.getPosition(self, ghostState)  
  
  
**def** add(x, y):  
 **return** (x[0] + y[0], x[1] + y[1])  
SAVE\_POSTSCRIPT = **False**POSTSCRIPT\_OUTPUT\_DIR = **'frames'**FRAME\_NUMBER = 0  
**import** os  
  
  
**def** saveFrame():  
 **global** SAVE\_POSTSCRIPT, FRAME\_NUMBER, POSTSCRIPT\_OUTPUT\_DIR  
 **if not** SAVE\_POSTSCRIPT:  
 **return  
 if not** os.path.exists(POSTSCRIPT\_OUTPUT\_DIR):  
 os.mkdir(POSTSCRIPT\_OUTPUT\_DIR)  
 name = os.path.join(POSTSCRIPT\_OUTPUT\_DIR, **'frame\_%08d.ps'** % FRAME\_NUMBER)  
 FRAME\_NUMBER += 1  
 writePostscript(name)

# **P**ROJE GERÇEKLEŞTİRİMİNDE YARARLANILAN KAYNAKLAR

* <http://aytugonan.cbu.edu.tr/>
* GitHub