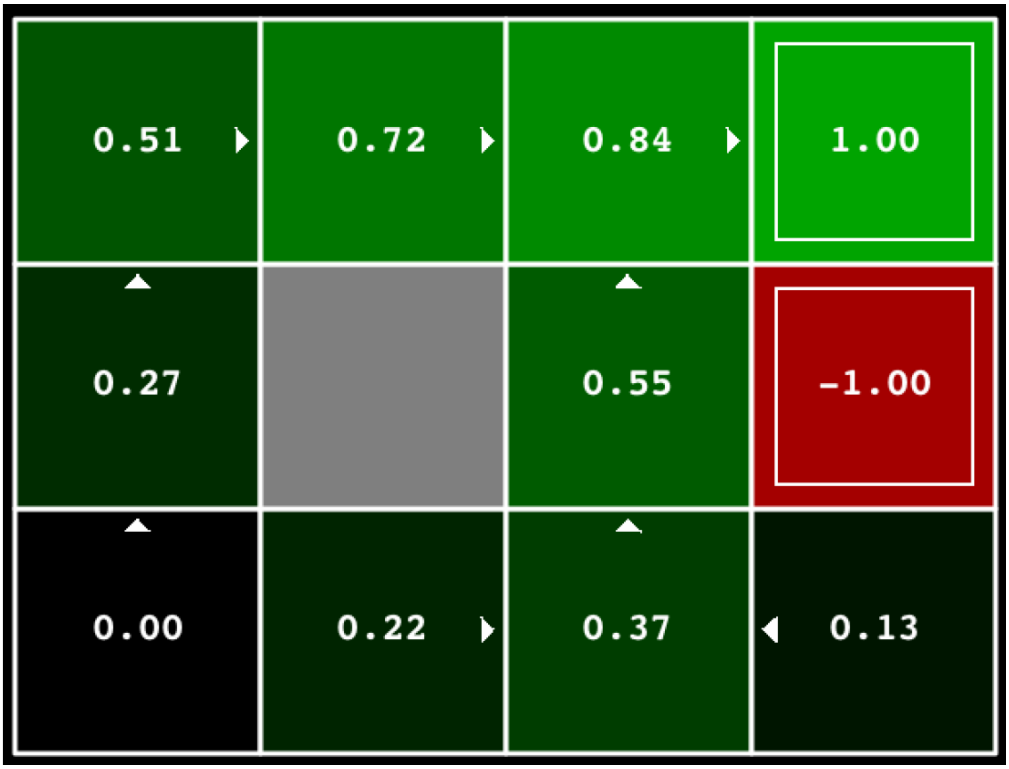
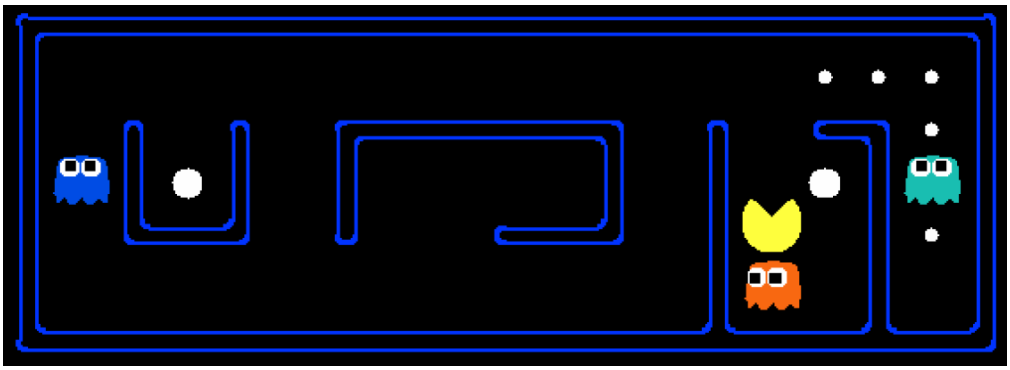
Laboration 2

MDPer, RL & GA





Sektionen för IT
Högskolan i Borås
501 90 Borås

Patrick Gabrielsson
patrick.gabrielsson@hb.se
Rum: L436

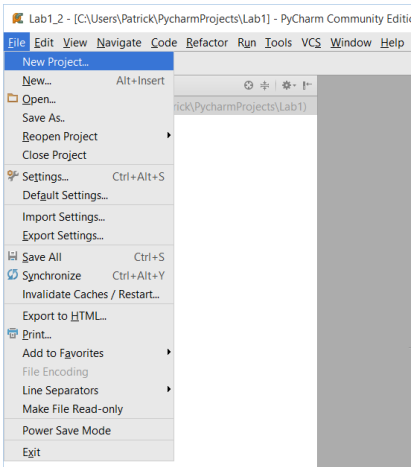
Innehåll

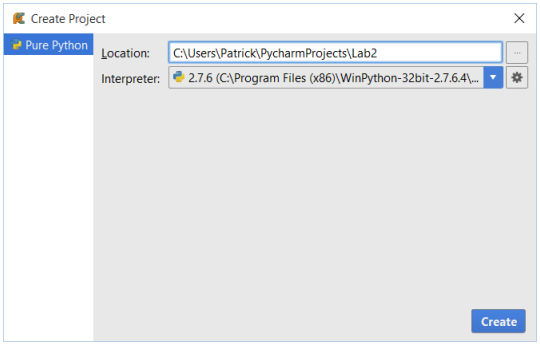
1 Inledning

Denna laborationen behandlar den *Genetiska Algoritmen*, *Markov Decision Processes (MDP)* och *Reinforcement Learning (RL)*, där *value iteration* och *Q-learning* algoritmerna kommer att implementeras. Agenter, som implementerar dessa algoritmerna, kommer att appliceras på *Gridworld* problemet och *Pacman* spelet. Dessutom kommer Q-learning agenten att testas på en robotkontroller (*Crawler*). Den genetiska algoritmen kommer att användas för att optimera ett antal funktioner.

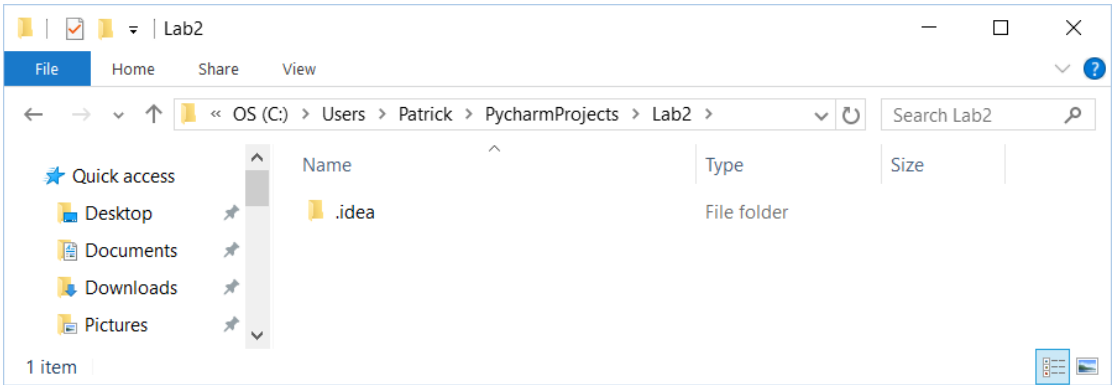
2 Installation och Introduktion

Starta PyCharm och skapa ett nytt projekt (t.ex. med namn "Lab2").

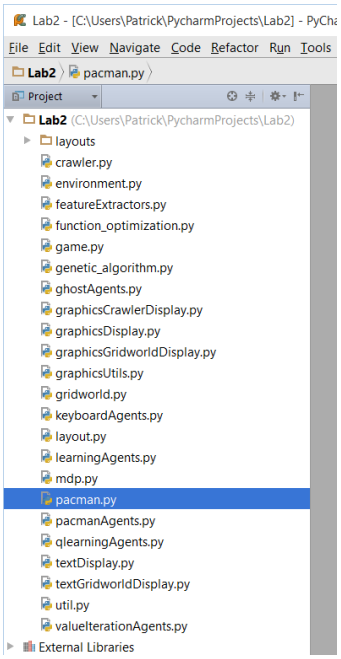


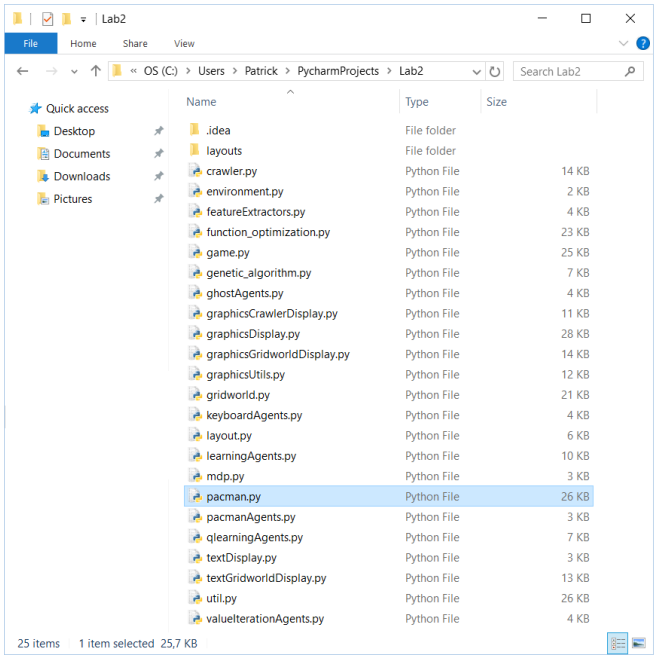


Ladda ner filen Lab2_filer.zip från PingPong och packa upp den i projektfoldern som ni skapade ovan.



Filen pacman.py skall nu finnas direkt under projektfoldern (i samma folder som katalogen ".idea" i nedanstående bild och inte i foldern Lab2_filer).



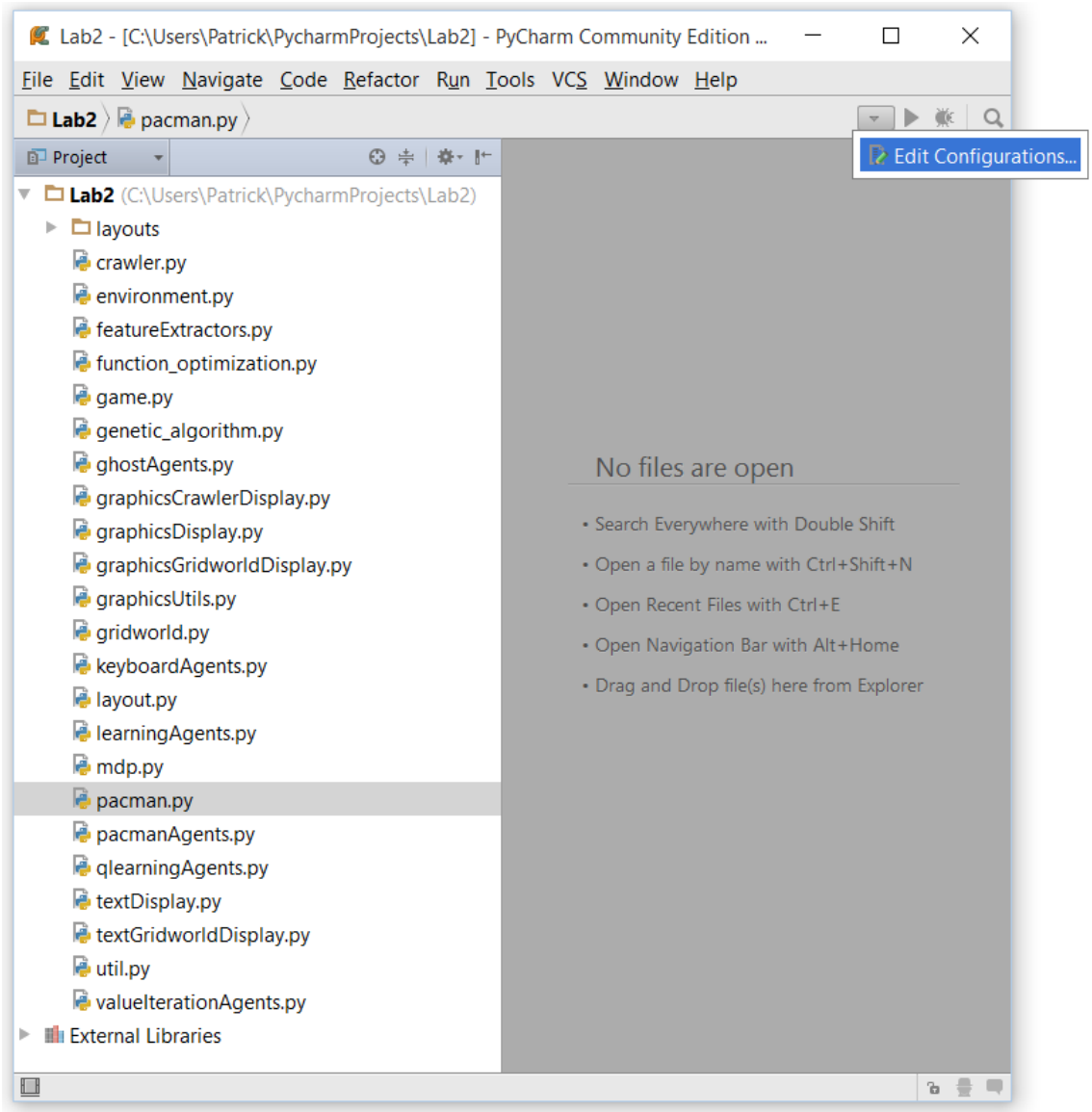


Följande filer ingår i projektet:

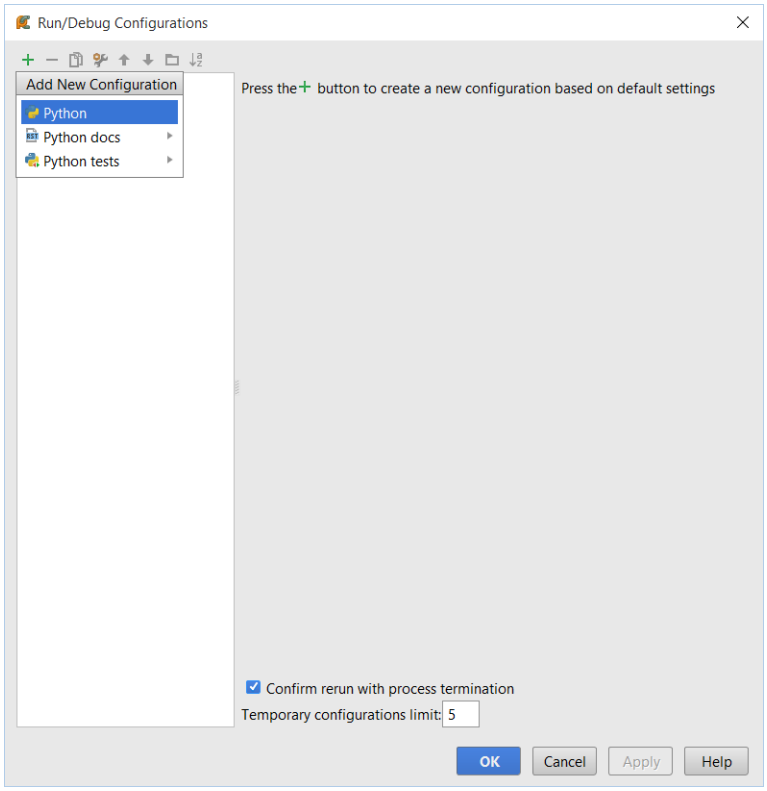
- <layouts> - folder som innehåller ett antal .lay filer (kartor "levels" för Pacman)
- crawler.py - innehåller kod för robotkontrollern (crawler)
- environment.py - innehåller en abstrakt klass för reinforcement learning miljöer
- *featureExtractors.py* - innehåller klasser för extrahering av (state,action) egenskaper
- function_optmization.py - innehåller kod för funktionsoptimering med GA
- game.py - innehåller logiken för Pacman-spelet
- **genetic_algorithm.py** - innehåller ett ramverk för den genetiska algoritmen (GA)
- ghostAgents.py - Agenter som kontrollerar spöken
- graphicsCrawlerDisplay.py - grafik för Crawler
- graphicsDisplay.py - grafik för Pacman
- graphicsGridworldDisplay.py - grafik för Gridworld
- graphicsUtils.py - innehåller support för grafik
- *gridworld.py* - innehåller implementeringen för Gridworld
- keyboardAgents.py - Tangentbordsinterface för att kontrollera Pacman
- layout.py - kod för att läsa layout filer och för att lagra deras innehåll
- *learningAgents.py* - innehåller klasserna ValueEstimationAgent och QLearningAgent som utgör basklasser för lärande agenter
- *mdp.py* - innehåller generella MDP metoder
- pacman.py - Huvudfilen som kör Pacman spel
- pacmanAgents.py - innehåller några simpla exempelagenter för Pacman
- **qlearningAgents.py** - Q-learning agenter för Gridworld, Crawler och Pacman
- textDisplay.py - ASCII grafik för Pacman
- textGridworldDisplay.py - ASCII grafik för Gridworld
- *util.py* - innehåller bland annat den användbara klassen "Counter" för Q-learning
- **valueIterationAgents.py** - En "value iteration" agent för lösning av kända MDPer

De enda filerna som skall modifieras och lämnas in är filerna **qlearningAgents.py**, **valueIterationAgents.py** och **genetic_algorithm.py** (markerade med **fet stil**). Filerna *featureExtractors.py*, *gridworld.py*, *learningAgents.py*, *mdp.py* och *util.py* (markerade med *kursiv stil*) kan vara bra att läsa igenom. Övriga filer behöver ni inte bry er om.

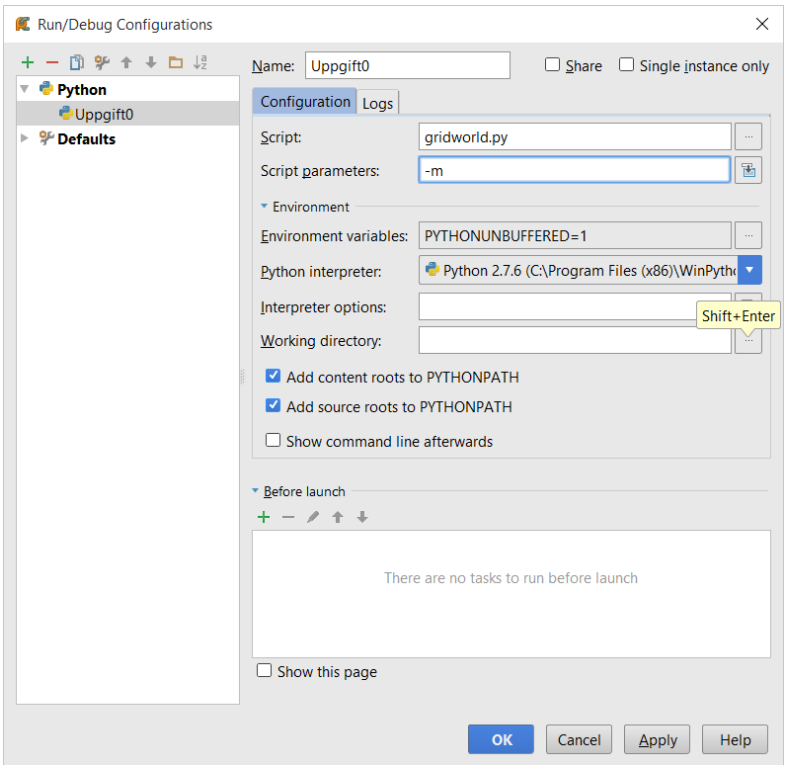
Klicka på pilen längst upp till höger i PyCharms GUI och välj **Edit Configurations** (se nedanstående bild).



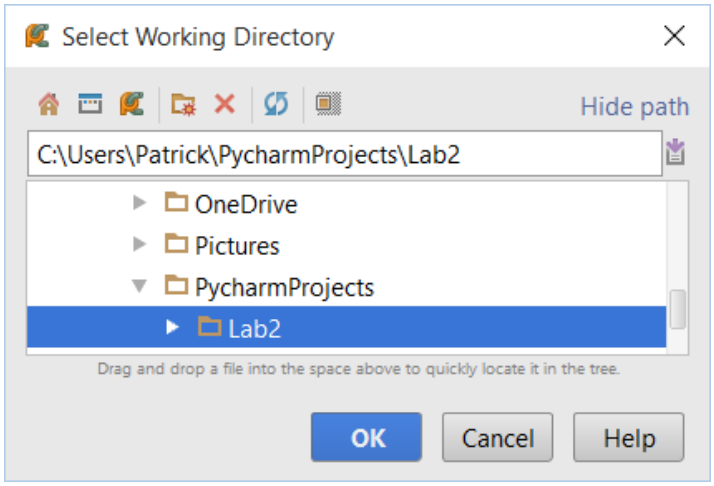
Klicka på "+"-knappen längst upp till vänster och välj **Python** från kontextmenyn (se nedanstående bild).



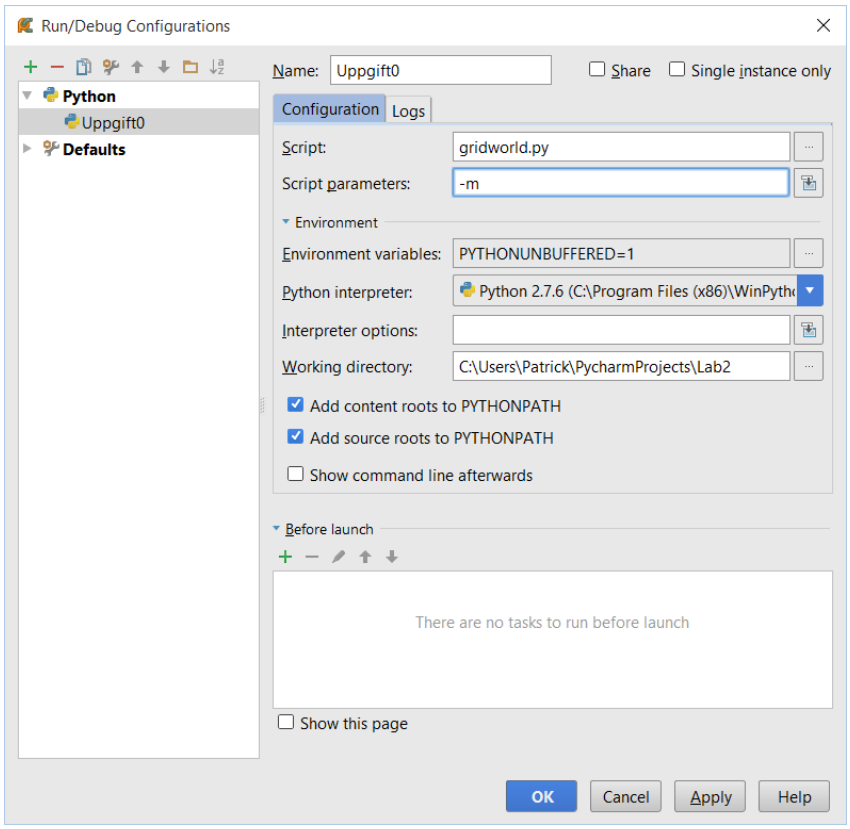
Fyll i "*Uppgift0*" i **Name** fältet, "*gridworld.py*" i **Script** fältet, "*-m*" i **Script parameters** fältet och klicka på "**...**"-knappen jämte **Working directory** fältet (se nedanstående bild).



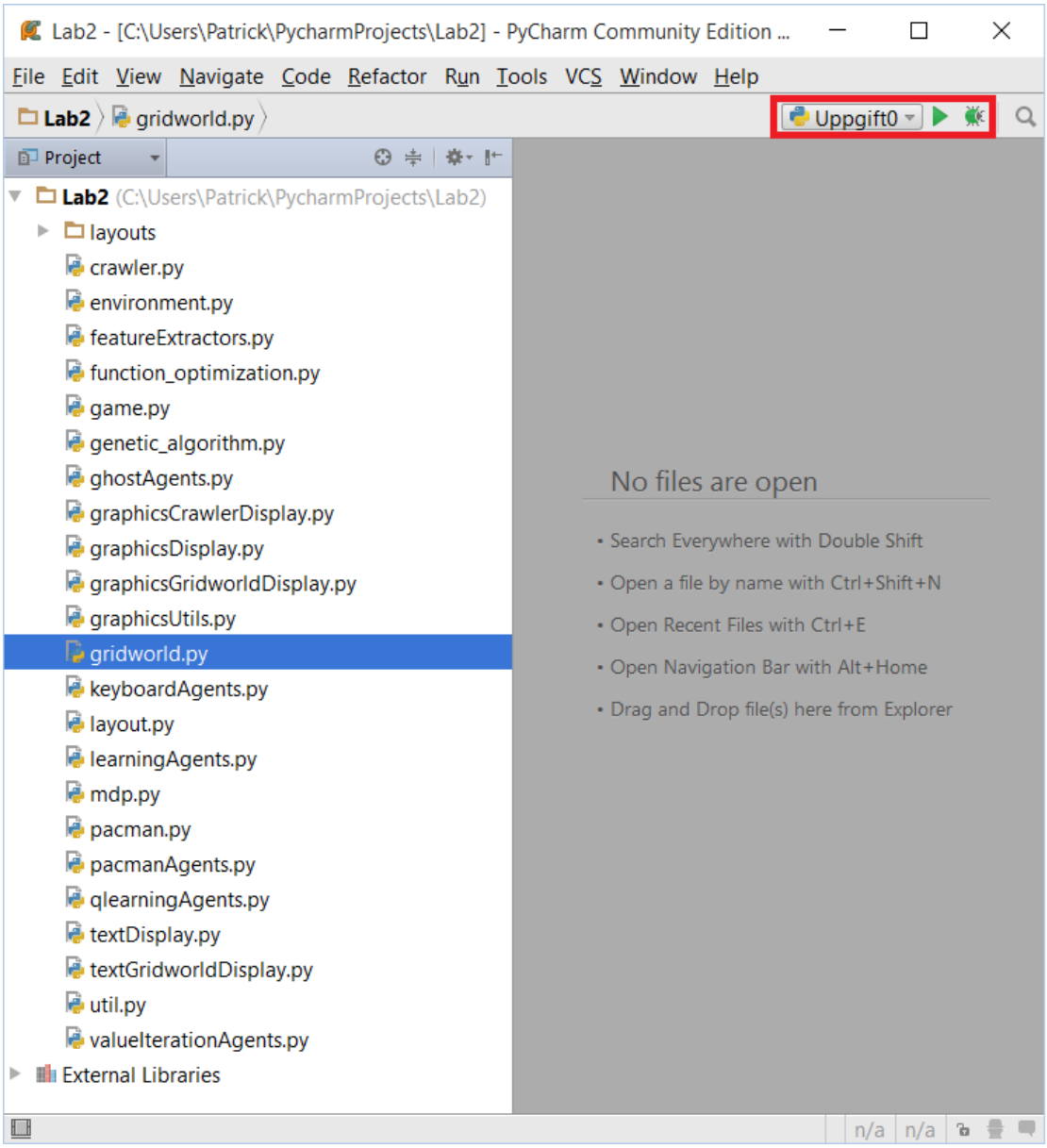
Se till så att din projektfolder är vald som din *Working Directory* och klicka på "**OK**"-knappen.



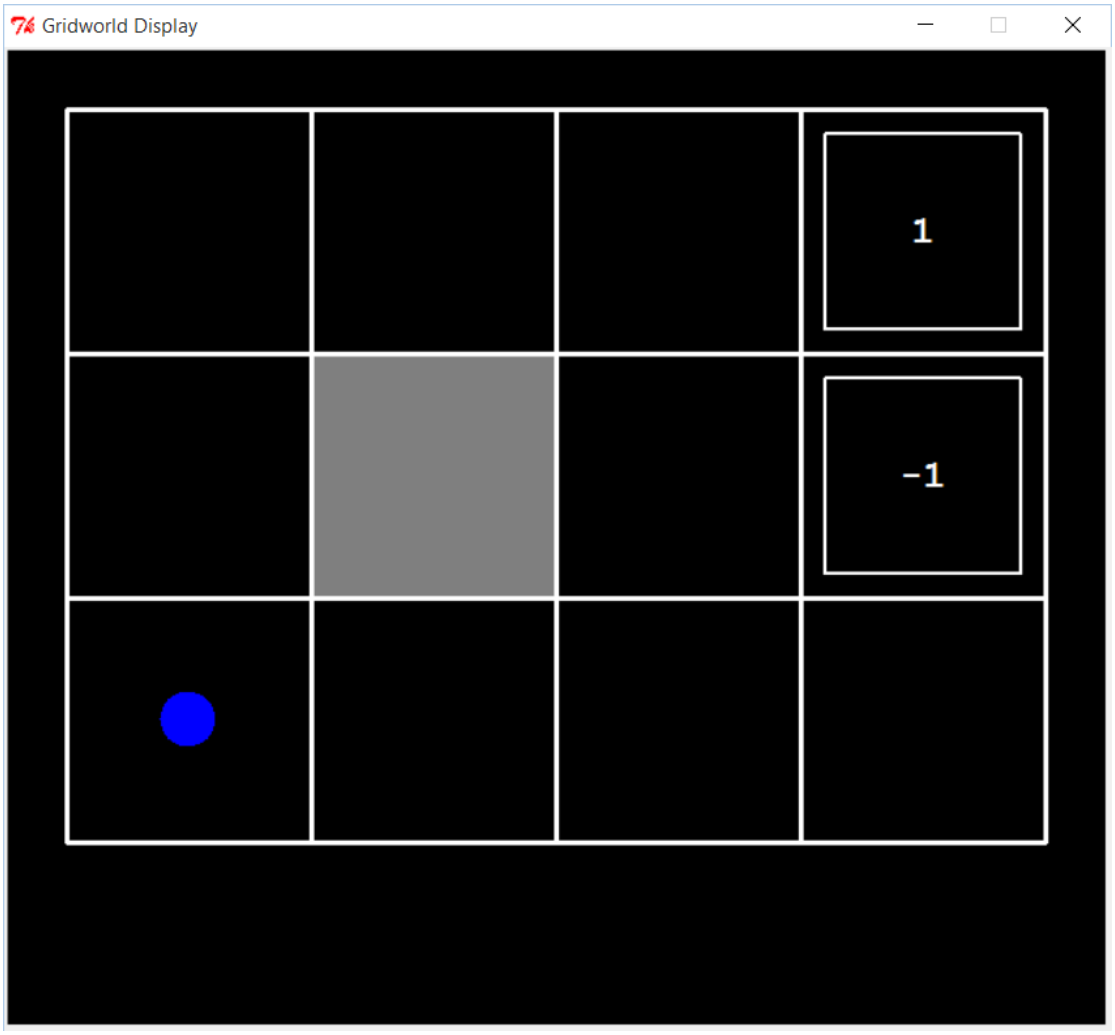
Klicka slutligen på "**OK**"-knappen i *Run/Debug Configurations* fönstret.



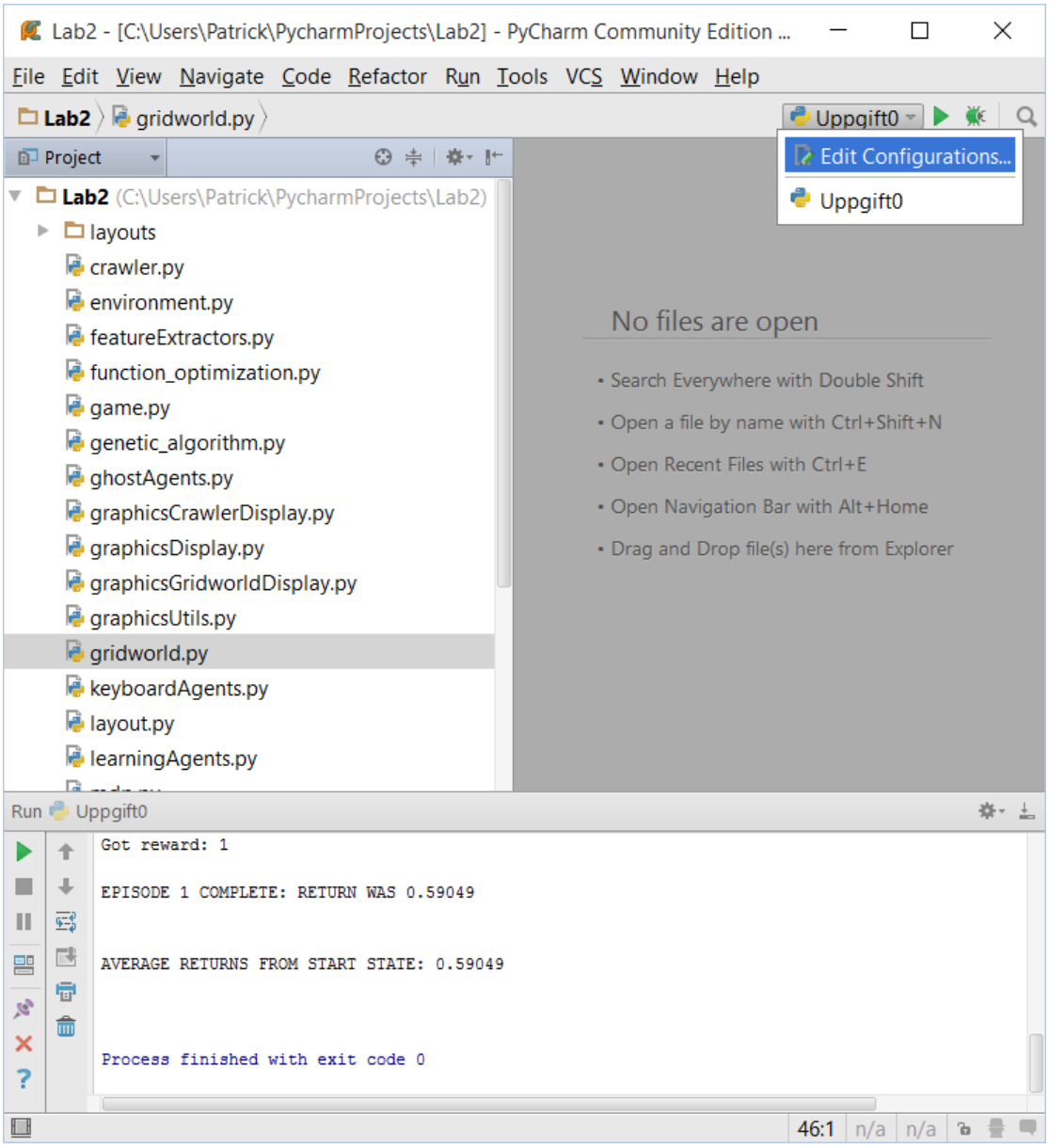
Ni har nu skapat en *konfiguration* som kör/debuggar Gridworld-problemet i normalt läge. Namnet på vald konfiguration ("Uppgift0") visas längst upp till vänster i PyCharms GUI (se nedanstående bild). För att testköra Gridworld, klicka på den gröna högerpilen jämte namnet på vald konfiguration. För att debugga Gridworld, klicka på den gröna "insekten" jämte den gröna högerpilen.



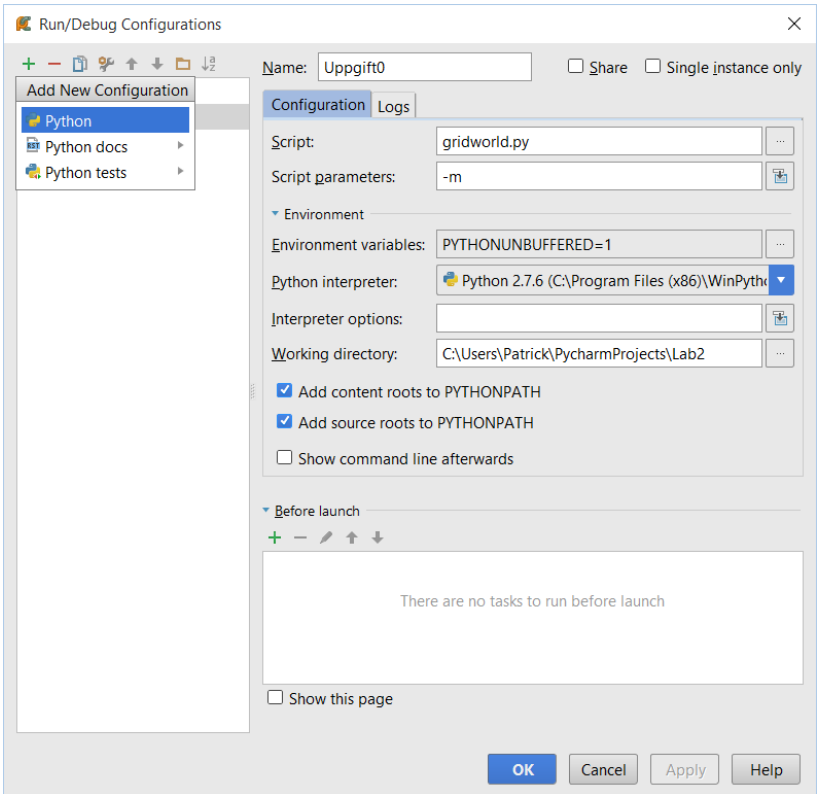
Gridworld applikationen körs nu i vanligt läge, där ni kan använda piltangenterma för att styra agenten (en blå prick). Precis som i Pacman brädet så har varje position i Gridworld brädet koordinaterna (x,y) med origo (0,0) längst ner till vänster. Observera att det finns en 20% chans att agenten går i en annan riktning än den önskade. Dessutom är de två tillstånden (1 samt -1) längst upp till höger i layouten speciella med avseende på att när agenten hamnar i en av dessa rutorna så finns det endast en giltig handling (i nästa tidsdiskreta steg) som tar agent till ett virtuellt tillstånd utanför brädet. Detta tillståndet utgör det terminala tillståndet. Observera också att med ovanstående Python konfiguration kommer *discountRate* att ha värdet 0.9 (vilket kan ändras med växeln -d) samt *livingReward* kommer att ha värdet 0 (vilket kan ändras med växeln -r). Stäng fönstret för att avsluta Gridworld applikationen.



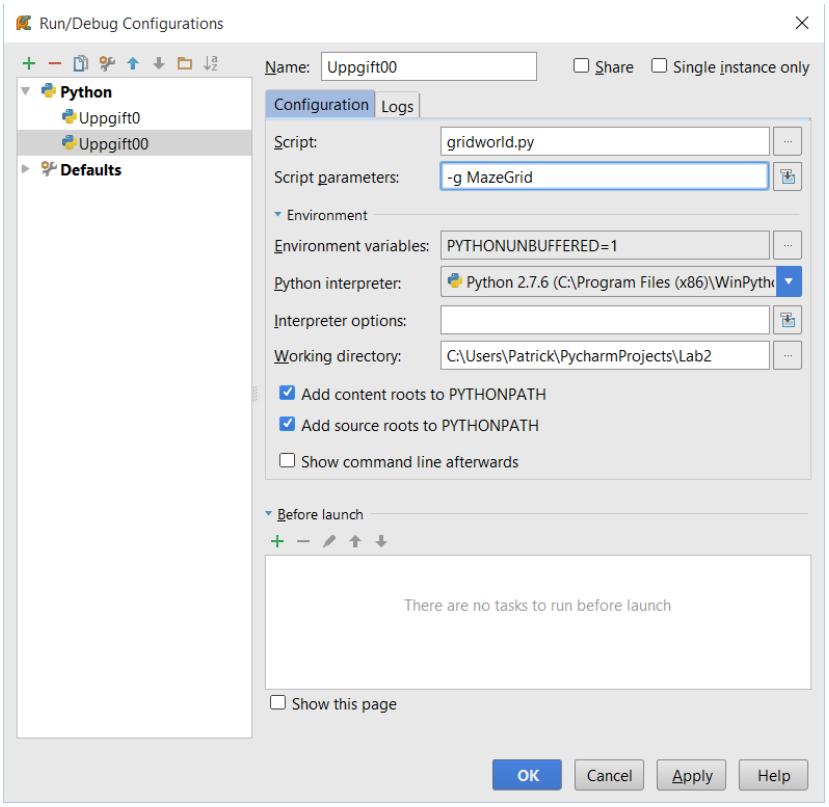
Klicka återigen på konfigurations-pilen längst upp till höger i PyCharms GUI och välj **Edit Configurations**.



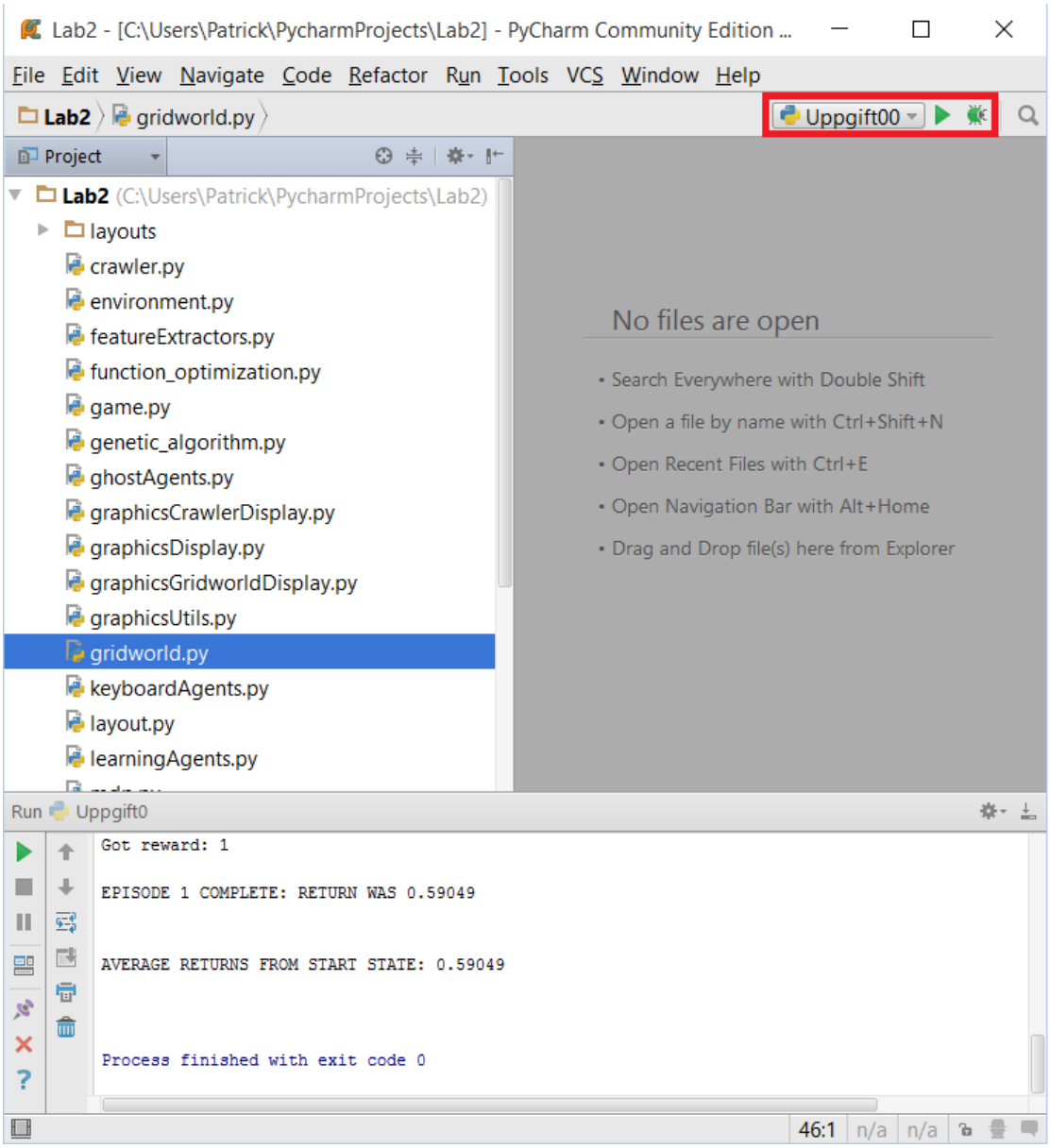
Klicka på "+"-knappen längst upp till höger och välj **Python**.



Fyll i "*Uppgift00*" i **Name** fältet, "*gridworld.py*" i **Script** fältet och "*-g MazeGrid*" i **Script parameters** fältet. Se också till så att projektfoldern är vald i **Working directory** fältet. Klicka sedan på "**OK**"-knappen.



Den nya konfigurationen ("Uppgift00") är nu förvald som aktiv konfiguration i konfigurations-fältet längst upp till höger i PyCharms GUI. Vi kan enkelt byta mellan olika konfigurationer genom att använda detta fältet. Klicka på kör-knappen eller debug-knappen för att testköra Gridworld med den nya konfigurationen.

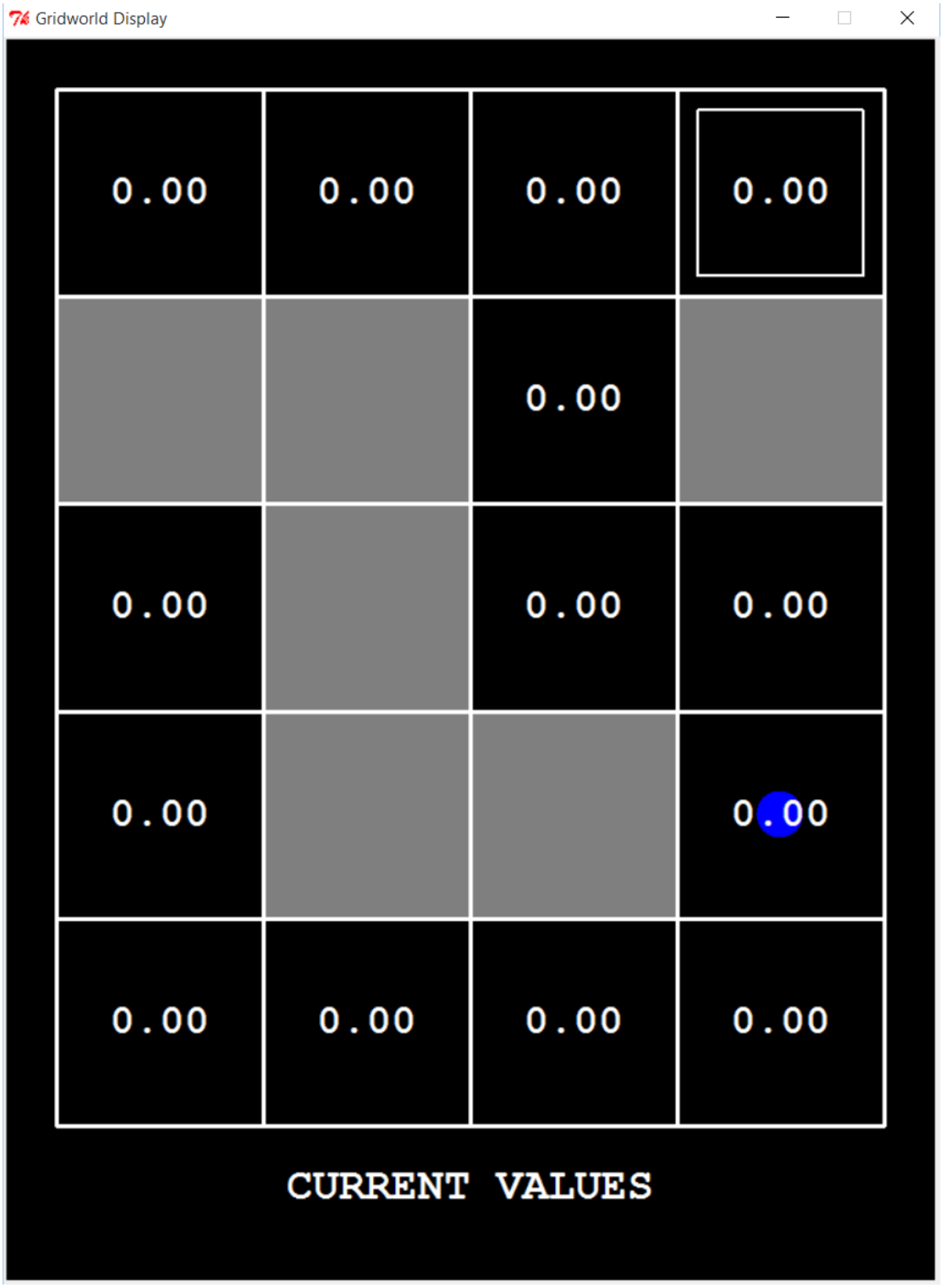


..

Den nya konfigurationen körde *gridworld.py* med följande växel:

-g MazeGrid

Denna växeln ser till så att layouten med namn *MazeGrid* (metoden **mazeGrid()** i filen **gridworld.py**) laddas. Dessutom kommer en default-agent att användas som rör sig slumpvist genom layouten tills en av de terminala tillstånden påträffas.



I nedanstående uppgifter skall MDP-lösande agenter samt lärande agenter skapas.

För varje deluppgift kommer en ny konfiguration att skapas med diverse växlar. Filerna *gridworld.py* och *pacman.py* kan acceptera ett flertal växlar, där varje växel kan anges med två bindessträck efterföljt av det fullständiga namnet på växeln alternativt med ett bindessträck efterföljt av en förkortning (en bokstav) av respektive växel. Exempelvis, är följande två rader ekvivalenta:

--grid=MazeGrid
-g MazeGrid

Nedan följer en beskrivning av samtliga växlar för *gridworld.py*:

- -n DISCOUNT, --discount=DISCOUNT
  diskonteringen (*gamma*) för framtida värden (default=0.9)
- -r R, --livingReward=R
  anger levnadsbelöningen för ett diskret tidssteg (default=0.0)
- -n P, --noise=P
  anger sannolikheten för att handlingar resulterar i en oavsiktlig riktning (default=0.2)
- -e E, --epsilon=E
  sannolikheten för att välja en slumpmässig handling i q-learning (default=0.3)
- -l P, --learningRate=P (default=0.5)
  inlärningshastigheten vid Temporal Difference (TD) inlärning
- -i K, --iterations=K
  antalet iterationer i value iteration (default=10)
- -k K, --episodes=K
  antalet epsioder som en MDP skall köras (default=1)
- -g G, --grid=G
  anger vilken layout (skiftlägeskänslig) som skall användas; BridgeGrid, CliffGrid, MazeGrid eller BookGrid (default)
- -w X, --windowSize=X
  anger bredden (antalet pixlar X) för varje cell i Gridworld grafiken (default=150)
- -a A, --agent=A
  anger agenttyp; 'value', 'q' eller 'random' (default)
- -t, --text
  anger att endast en ASCII-text display skall användas (istället för en GUI)
- -p, --pause
  anger att GUIt skall pausas efter varje diskret tidssteg då MDPn körs
- -q, --quiet
  anger att utskriften av varje inlärningsepisod är avslagen
- -s S, --speed=S
  Anger animeringshastigheten, S>1.0 snabbare, 0.0<S<1.0 långsammare (default=1.0)
- -m, --manual
  anger att agenten skall kontrolleras manuellt (via piltangenterna på tangentbordet)
- -v, --valueSteps
  anger att varje steg i "value iteration" skall visas

Nedan följer en beskrivning av samtliga växlar för *pacman.py*:

- -n GAMES, --numGames=GAMES
  Antalet spel (GAMES) som skall köras (default=1).
- -l LAYOUT_FILE, --layout=LAYOUT_FILE
  Anger vilken layout fil (LAYOUT_FILE) som skall användas (default=mediumClassic)
- -p TYPE, --pacman=TYPE
  anger agenttypen (i pacmanAgents.py) som skall användas (default:KeyboardAgent)
- -t, --textGraphics Visar endast text som output
- -q, --quietTextGraphics Skapar minsta möjliga output och ingen grafik
- -g TYPE, --ghosts=TYPE
  spökagenttypen TYPE (i ghostAgents.py) som skall användas (default=RandomGhost)
- -k NUMGHOSTS, --numghosts=NUMGHOSTS
  Max antal spöken (default=4)
- -z ZOOM, --zoom=ZOOM Zoomar storleken på spelfönstret (default=1.0)
- -f, --fixRandomSeed Fixerar fröet till slumptalsgeneratorn (så att samma spel spelas)
- -r, --recordActions Skriver spelhistorik till en fil
- --replay=GAMETOREPLAY Ett inspelat spel som skall spelas upp
- -a AGENTARGS, --agentArgs=AGENTARGS
  Kommaseparerad lista med namn/värde-par som skall skickas till agenten t.ex. "opt1=val1,opt2,opt3=val3"
- -x NUMTRAINING, --numTraining=NUMTRAINING
  Hur många episoder som utgör träningsepisoder (default=0)
- --frameTime=FRAMETIME
  Tidsfördröjning mellan " frames" (default=0.1)
- -c, --catchExceptions
  Slår på felhantering och time-out:er under spel
- --timeout=TIMEOUT Max tid en agent har på sig för beräkningar (default=30)

3 Value Iteration

I filen **learningAgents.py** hittar ni klassen **ValueEstimationAgent** som ärver från klassen **Agent** (i filen **game.py**). **ValueEstimationAgent** utgör en abstrakt basklass för en agent som kan beräkna tillståndsvärden *V(s)*, q-värden *Q(s,a)* och policyn *Π(s)*. Som bekant accepterar *V(s)* ett tillstånd *s* (som i Gridworld och Pacman utgörs av tupler (x,y) med positioner på ett två-dimensionellt bräde), och returnerar ett reellt tal som representerar hur mycket tillståndet är värt, dvs *nyttan* för agenten att befinna sig i just det tillståndet. På motsvarande sätt accepterar *Q(s,a)* ett tillstånd *s* och en handling *a* i det tillståndet (exempel på handlingar i Gridworld och Pacman är *north*, *south*, *east* och *west*) och returnerar ett reellt tal som representerar hur mycket handlingen i tillståndet är värt, dvs *nyttan* för agenten att utföra just den handlingen i just det tillståndet. En policy *Π(s)* accepterar ett tillstånd *s* och returnerar den handling *a* med högst värde (störst nytta) i just det tillståndet. Klassen innehåller en konstruktor **__init__()** och de fyra metoderna **getQValue()**, **getValue()**, **getPolicy()** samt **getAction()**.

Konstruktorn accepterar fyra inparametrar; *alpha*, *epsilon*, *gamma* och *numTraining* med respektive defaultvärden *1.0*, *0.05*, *0.8* och *10*. Samtliga fyra parametrar används då en agent lär sig en MDP (*Markov Decision Process*), där *alpha* är inlärningshastigheten, *epsilon* är utforskningssannolikheten, *gamma* är "discount rate" (dvs hur "närsynt" eller "långsynt" agenten är) och *numTraining* är antalet träningsepisoder. Dessa fyra parametrarna lagras helt enkelt i instansvariabler med samma namn i konstruktorn (**OBS!** Inparametern *gamma* lagras dock istället i en instansvariabel med namnet *discount*.)

Metoden **getQValue()** accepterar ett tillstånd *s* samt en handling *a* och meningen är att en ärvd klass skall returnera det motsvarande q-värdet *Q(s,a)*.

Metoden **getValue()** accepterar ett tillstånd *s* och meningen är att en ärvd klass skall returnera det motsvarande tillståndsvärdet *V(s)*, dvs genom att välja det största *Q(s,a)* värdet bland alla giltiga handlingar *a* i tillståndet *s*.

Metoden **getPolicy()** accepterar ett tillstånd *s* och meningen är att en ärvd klass skall returnera motsvarande handling *a*, dvs genom att välja handlingen *a*, bland alla giltiga handlingar i tillståndet *s*, som ger det största *Q(s,a)* värdet. Observera att om en *ϵ-greedy* policy implementeras så kommer inte alltid handlingen *a*, med tillhörande största *Q(s,a)* värde, att returneras.

Metoden **getAction()** är som **getPolicy()** fast returnerar alltid handlingen *a* som ger det största *Q(s,a)* värdet.

I filen **valueIterationAgents.py** finns klassen **ValueIterationAgent** som ärver från klassen **ValueEstimationAgent** (som beskrevs ovan) och implementerar de fyra abstrakta metoderna **getQValue()**, **getValue()**, **getPolicy()** samt **getAction()**. Observera dock att **getQValue()** anropar metoden **computeQValueFromValues()** samt **getPolicy()** och **getAction()** anropar båda metoden **computeActionFromValues()**. Dessa två metoderna är inte implementerade. Dessutom saknas en del kod i konstruktorn **__init__()**.

3.1 ValueIterationAgent

Implementera *value iteration* algoritmen i klassen **ValueIterationAgent** (i filen **valueIterationAgents.py**) genom att komplettera konstruktorn **__init__()** samt de två metoderna **computeQValueFromValues()** och **computeActionFromValues()**.

Konstruktorn **__init__()** accepterar tre inparametrar; *mdp*, *discount* och *iterations*. Parametern *mdp* representerar ett MDP problem som agenten skall lösa, *discount* är value iteration algoritmens *gamma* parameter (med defaultvärdet 0.9) och *iterations* anger antalet iterationer för value iteration algoritmen (med defaultvärdet 100). I konstruktorn skall ni implementera själva *value iteration* algoritmen som iterativt beräknar värdet för varje tillstånd (i så många iterationer som anges av inparametern *iterations*) och lagrar samtliga tillståndsvärden *V(s)* i klassvariabeln *self.values*, som består av en Python *dictionary* med elementen {state: value} - ett för varje tillstånd. Det ni skall beräkna i varje iteration (för varje tillstånd *s*) motsvaras alltså av uppdateringsregeln för value iteration enligt nedan (som beräknar *k*-stegs estimerade värden för de optimala tillståndsvärdena):

$$V_{k+1}(s) \leftarrow \max_a \sum_{s\prime} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

Ovanstående uppdateringsregel är ekvivalent med:

$$V_{k+1}(s) \leftarrow \max_a Q(s,a)$$

där

$$Q(s,a) = \sum_{s\prime} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

För att kunna implementera uppdateringsregeln för value iteration måste ni därför implementera beräkningen av *Q(s,a)* ovan i metoden **computeQValueFromValues()**. Denna metoden accepterar de båda inparametrarna *state* och *action*, precis som metoden **getQValue()**. Eftersom **getQValue()** endast gör ett direkt anrop till **computeQValueFromValues()**, kan ni helt enkelt anropa **getQValue()** ifrån konstruktorn efter att ni har implementerat **computeQValueFromValues()**.

Slutligen, efter att agenten har beräknat samtliga tillståndsvärden med value iteration algoritmen, så måste angenten kunna utföra motsvarande handlingar i varje tillstånd som löser MDP problemet optimalt. Detta görs genom anrop till **getAction()** från Gridworld applikationens **__main__** kodblock längst ner i filen **gridworld.py**. Dessutom görs ett anrop till **getPolicy()** från metoden **displayValues()** i filen **graphicsGridworldDisplay.py** för att kunna visa Gridworld agentens policy (pilarna i varje ruta) grafiskt. Både **getAction()** och **getPolicy()** gör endast ett direkt anrop till metoden **computeActionFromValues()**. Metoden **computeActionFromValues()** accepterar den enda inparametern *state* och måste alltså returnera vilken handling som agenten skall utföra i just det tillståndet, dvs metoden skall implementera **policy extraction** algoritmen enligt nedan:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

som motsvarar

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

där * indikerar ett optimalt värde (dock är denna notationen inte helt korrekt eftersom vi beräknar estimerade värden och inte matematiskt optimala värden).

Från denna sista ekvationen ser vi att även här måste vi anropa metoden **getQValue()** från **computeActionFromValues()**.

OBS! Ni skall implementera *batch-versionen* av value iteration algoritmen i konstruktorn **__init__()**. Detta betyder helt enkelt att ni behöver två stycken *Python dictionary* objekt; ett för att hålla reda på samtliga tillståndsvärden *V(s)* för iteration *k* samt ett annat för att hålla reda på samtliga tillståndsvärden *V(s)* för iteration *k+1* (dock skall ni använda klassen **Counter** i filen **utils.py** istället för en *Python dictionary*). Ni vill alltså använda samtliga gamla tillståndsvärden när ni beräknar nya tillståndsvärden och sedan byta ut det gamla *utils.Counter* objektet mot det nya sist i varje iteration. Skillnaden mellan en *batch update* och en *in-place update* är alltså att datastrukturerna för $V_{k+1}(s)$ och $V_k(s)$ utgör två olika instanser i en *batch update*, medans de utgör samma instans i en *in-place update* (se kapitel 4.1 i referensboken http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node41.html). Man kan visa att båda varianterna konvergerar. Tänk också på att om ett tillstånd är ett terminaltillstånd (state=TERMINAL_STATE) så är *V(s)*=0.

OBS! Dessutom måste ni hantera fallet då ett tillstånd inte har några giltiga handlingar (vilket tillståndsvärde *V(s)* skall ni använda i så fall, dvs vad betyder det för framtida belöningar när ett tillstånd inte har några giltiga handlingar?).

Från **mdp** objektet som skickas in via konstruktorn **__init__()** kan följande information hämtas:

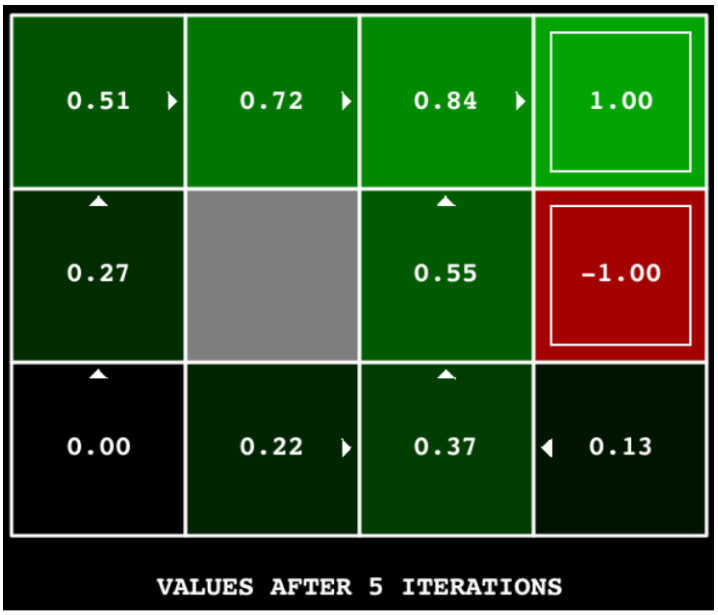
- *mdp.getStates()* returnerar en lista med samtliga tillstånd
- *mdp.getPossibleActions(state)* returnerar en lista med giltiga handlingar i *state*
- *mdp.getTransitionStatesAndProbs(state, action)* returnerar paret *{s',p}* där *p* anger sannolikheten att hamna i tillståndet *s'*
- *mdp.getReward(state, action, nextState)* returnerar den omedelbara belöningen (*immediate reward*) då handlingen *action* utförs i tillståndet *state* varpå agenten hamnar i tillståndet *nextState*.
- *mdp.isTerminal(state)* returnerar *True* om *state* är ett terminaltillstånd

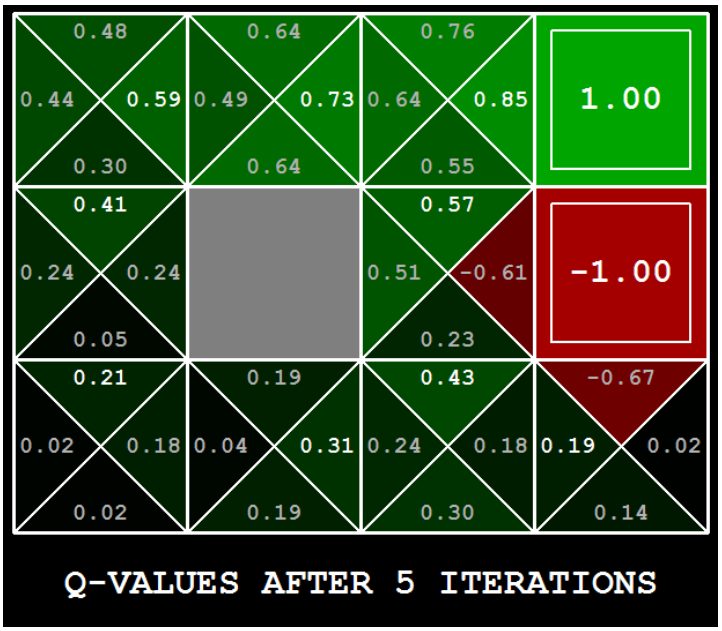
Testa agenten med nedanstående Python konfiguration (10 rundor, med 100 iterationer var, med *BookGrid* layouten). När Gridworld först blir synlig, visas tillståndsvärdet *V(s)* i varje ruta tillsammans med policyn *Π(s)*. Tryck därefter på valfri tangent för att byta till nästa vy som visar q-värdet *Q(s,a)* för varje kvadrant i varje ruta. Tryck slutligen återigen på valfri tangent för att låta agenten följa policyn.

- gridworld.py -g BookGrid -a value -i 100 -k 10

Med en korrekt implementerad value iteration agent skall era tillståndsvärden (värdet i varje ruta), era q-värden (värdet i varje kvadrant i varje ruta) samt eran policy (pilarna i varje ruta) se ut enligt figurerna nedan om ni kör följande Python konfiguration (dvs efter 5 iterationer med *BookGrid* layouten):

- gridworld.py -g BookGrid -a value -i 5





3.2 BridgeGrid

När ni har en fungerande *value iteration* agent skall vi undersöka hur de båda parametrarna *discount* samt *noise* påverkar agentens optimala policy för *BridgeGrid* problemet.

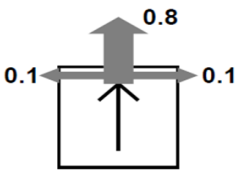
BridgeGrid layouten visas i nedanstående bild och innehåller två positiva samt tio negativa terminaltillstånd. Det ena positiva terminaltillståndet (längst till vänster) har en liten belöning (+1.00) medans det andra positiva terminaltillståndet (längst till höger) har en stor belöning (+10.00). Mellan de två positiva terminaltillstånden finns en "bro" som omges av fem negativa terminaltillstånd på vardera sida med mycket stora negativa belöningar (-100.00). Levnadskostnaden för BridgeGrid problemet är 0, dvs varje omedelbar belöning (*immediate reward*) är 0 (förrutom för de 12 terminaltillstånden förståss). Agenten börjar i tillståndet omedelbart till höger om det positiva terminaltillståndet med belöningen +1.00, dvs i rutan som visar talet -17.28 i nedanstående bild.

Med nedanstående Python konfiguration kommer den optimala policyn **inte** att korsa bron och lämna brädet via det positiva terminaltillståndet med belöningen +10.00:

- gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2

Ovanstående konfiguration använder en *discount* (*gamma* värde) på 0.9 med en stokastisk handlingsprofil (*noise*) där sannolikheten att gå i en annan riktning än den önskade är 0.2. Med en levnadskostad på 0, fundera på hur de två parametrarna *discount* och *noise* påverkar uppdateringsformeln i value iteration algoritmen.

Discount är samma sak som *gamma "γ"* parametern i uppdateringsformeln medans *noise* parametern fördelar sin sannolikhet över de två vinkelräta riktningarna i förhållande till den önskade riktningen i *övergångsfuntionen* (*transition function*) "*T(s,a,s')*" enligt nedanstående figur (här med *noise* = 0.2).



Kom ihåg att value iteration algoritmens uppdateringsformel

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

är ekvivalent med

$$V_{k+1}(s) \leftarrow \max_a Q(s,a)$$

där

$$Q(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

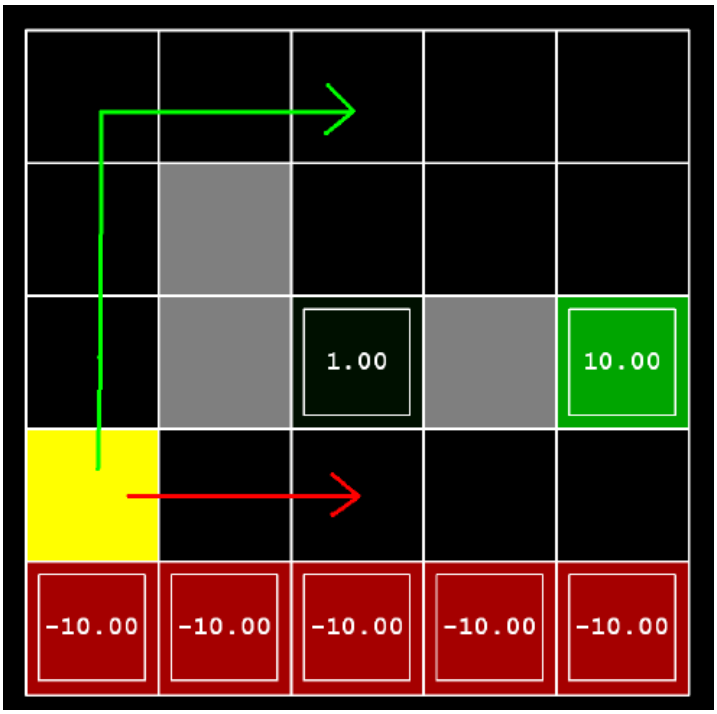
Fundera först på hur de två parametrarna kommer att påverka den optimala policyn (som ju väljer handling enligt *policy extraction* algoritmen som visades i föregående uppgift) för BridgeGrid problemet. Ändra därefter endast en av parametrarna (*discount* **eller** *noise*) i ovanstående konfiguration så att agentens policy korsar bron och lämnar brädet via terminaltillståndet +10.00 längst till höger. Med en korrekta policy skall eran lösning likna figuren nedan.

Fyll i era parametervärden för *discount* och *noise* i laborationsrapporten för laboration 2 (*Rapport_Laboration_2.docx*). Förklara även varför era parameterinställningar leder till önskad policy för agenten.

3.3 DiscountGrid

En annan viktig parameter, utöver *discount* och *noise*, som påverkar value iteration agentens policy är levnadskostnaden (*livingReward*), dvs den omedelbara belöningen (*immediate reward*) för varje tillstånd förrutom terminaltillstånden. Undersök hur de tre parametrarna (*discount*, *noise* och *livingReward*) påverkar agentens optimala policy för *DiscountGrid* problemet.

DiscountGrid layouten visas i nedanstående bild och innehåller två positiva samt fem negativa terminaltillstånd. Det ena positiva terminaltillståndet (i mitten) har en liten belöning (+1.00) medans det andra positiva terminaltillståndet (längst till höger) har en stor belöning (+10.00). Längst den nedersta raden finns fem negativa terminaltillstånd med stora negativa belöningar (-10.00). Agenten börjar i det gula tillståndet längst ner till vänster (se nedanstående figur). Dessutom finns det tre gråa rutor som innehåller väggar. För att nå terminaltillstånden kan agenten välja att gå den längre säkra vägen (grön pil) som undviker den nedersta raden, eller så kan agenten välja att gå den kortare osäkra vägen (röd pil) utefter den nedersta raden. Att denna vägen är osäker innebär att agenten riskerar att hamna i ett terminaltillstånd med stor negativ belöning.



Nedanstående Python konfiguration visar hur ni kan skicka med parametervärden för *discount*, *noise* samt *livingReward* till value iteration agenten:

- gridworld.py -a value -i 100 -g DiscountGrid --discount 0.0 --noise 0.0 --livingReward 0.0

Fundera på hur de tre parametrarna *discount*, *noise* och *livingReward* påverkar uppdateringsformeln i value iteration algoritmen. *Discount* är samma sak som *gamma* "γ" parametern i uppdateringsformeln, *noise* parametern påverkar *övergångssannolikheten* (*transition probability*) "*T(s,a,s')*" och *livingReward* parametern är samma sak som *immediate reward*, dvs *R(s,a,s')*.

Nedan anges återigen value iteration algoritmens uppdateringsformel

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

som är ekvivalent med

$$V_{k+1}(s) \leftarrow \max_a Q(s,a)$$

där

$$Q(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

Efter att ni har funderat på hur de tre parametrarna kommer att påverka den optimala policyn (som ju väljer handling enligt *policy extraction* algoritmen som visades i kapitel 3.1) för DiscountGrid problemet, försök att komma på Pyhton konfigurationer som leder till följande policyn:

a) Agenten föredrar att gå till terminaltillståndet +1.00 via den röda pilen.
b) Agenten föredrar att gå till terminaltillståndet +1.00 via den gröna pilen.
c) Agenten föredrar att gå till terminaltillståndet +10.00 via den röda pilen.
d) Agenten föredrar att gå till terminaltillståndet +10.00 via den gröna pilen.
e) Agenten föredrar att undvika alla terminaltillstånd, dvs episoden terminerar aldrig.

För att kontrollera eran policy kan ni följa pilarna i den grafiska framställningen av DiscountGrid brädet. Exempelvist skall policyn för deluppgift c) gå fyra steg österut och sedan ett steg norrut för att gå till terminaltillståndet +10.00 via den röda pilen (se nedanstående bild).

Fyll i era parametervärden för *discount*, *noise* och *livingReward* iför deluppgift a) - e) i laborationsrapporten för laboration 2 (Rapport_Laboration_2.docx). Förklara även varför era parameterinställningar leder till önskad policy.

4 Q-Learning (Reinforcement Learning)

Value iteration agenten har en komplett MDP modell, dvs känner till övergångsfunktionen *T(s,a,s')* och belöningsfunktionen *R(s,a,s')*. Därför behöver agenten inte lära sig något om sin omgivning (allt är redan känt) utan den kan använda value iteration algoritmen för att beräkna en optimal policy innan den ens börjar interagera med sin omgivning. För verkliga problem är dock både *T(s,a,s')* och *R(s,a,s')* okända, dvs agenten måste själv lära sig hur sin omgivning fungerar och hur olika handlingar påverkar omgivningen genom att interagera med omgivningen. Sådana agenter kallas för lärande agenter, där Q-learning agenten är en viss typ av lärande agent som implemenetrar Q-learning algoritmen.

I filen **learningAgents.py** finns den abstrakta klassen **ReinforcementAgent** som ärver från **ValueEstimationAgent**. Förrutom de ärvda abstrakta metoderna **getQValue()**, **getValue()**, **getPolicy()** och **getAction()** så definierar klassen ytterligare en abstrakt metod **update()**. De första fyra ärvda abstrakta metoderna förklarades utförligt i uppgift 3. Metoden **update()** accepterar de fyra inparametrarna *state*, *action*, *nextState* samt *reward* och meningen är att en ävrande klass skall använda dessa fyra parametrarna för att uppdatera sina Q-värden. Metoden **update()** kommer att anropas automatiskt av metoden **observeTransition()** i samma klass då en övergång observeras, dvs då agenten utför handlingen *action* i tillståndet *state* varpå agenten hamnar i tillståndet *nextState* och erhåller belöningen *reward* där. Förrutom dessa metoder innehåller klassen bland annat metoderna **getLegalActions()**, **startEpisode()**, **stopEpisode() isInTraining()**, **isInTesting()** samt konstruktorn **__init__()**. Den enda metoden som skall anropas direkt från en ärvande klass är **getLegalActions()** som accepterar ett tillstånd *state* och returnerar en lista med giltiga handlingar för det tillståndet. I konstruktorn ser ni att defaultvärden sätts för parametrarna *numTraining* (default=100), *epsilon* (default=0.5), *alpha* (default=0.5) och *gamma* (default=1). Parametern *numTraining* anger under hur många perioder som agenten skall lära sig, *epsilon* anger sannolikheten för att agenten skall utföra en slumpvis handling i ett tillstånd (dvs ϵ*-greedy* värdet), *alpha* är inlärningshastigheten (*learning rate*) och *gamma* är *discount rate* i q-learning algoritmen.

Längst upp i filen **qlearningAgents.py** finns klassen **QLearningAgent** definierad. Denna klassen ärver från **ReinforcementAgent** och måste alltså implementera de fem abstrakta metoderna **getQValue()**, **getValue()**, **getPolicy()**, **getAction()** och **update()**. Observera dock att **getValue()** redan är implementerad och endast gör ett direkt anrop till metoden **computeValueFromQValues()**. Metoden **getPolicy()** är också redan implementerad och gör endast ett direkt anrop till metoden **computeActionFromQValues()**. Med andra ord så måste metoderna **getQValue()**, **computeValueFromQValues()**,**computeActionFromQValues()**, **getAction()** och **update()** implementeras. Dessutom kan ni använda konstruktorn **__init__()** för att skapa ytterligare instansattribut, t.ex. någon lämplig datastruktur som ni kan lagra era *Q(s,a)*-värden i.

4.1 QLearningAgent

Implementera *q-learning* algoritmen i klassen **QLearningAgent** (i filen **qlearningAgents.py**) genom att komplettera metoderna **update()**, **computeValueFromQValues()**, **getQValue()**, **computeActionFromQValues()** samt konstruktorn **__init__()**.

Konstruktorn **__init__()** accepterar ett variabelt antal inparametrar som endast skickas vidare till superklassen **ReinforcementAgent**. Konstruktorn är ett lämpligt ställe att initiera en datastruktur för att lagra *Q(s,a)*-värden. Observera att inget arbete utförs i konstruktorn eftersom agenten måste lära sig genom *trail-and-error*, dvs genom att interagera med sin omgivning (jämfört detta med **ValueIterationAgent** agenten där allt arbete utfördes i konstruktorn).

Metoden **getQValue()** accepterar två inparametrar; ett tillstånd *state* och en handling *action* i det tillståndet. Metoden skall returnera q-värdet *Q(state, action)*, dock om tillståndet aldrig har påträffats av agenten innan skall värdet 0 returneras.

Metoden **computeValueFromQValues()** accepterar en inparameter; ett tillstånd *state*. Metoden skall returnera det största q-värdet *Q(state, action)* bland alla giltiga handlingar *actions* som kan utföras i tillståndet *state*. Dock om inga giltiga handlingar finns för tillståndet (t.ex. för ett terminaltillstånd) skall värdet 0 returneras. Här kan det vara lämpligt att använda superklassens metod *getLegalActions(state)*.

Metoden **computeActionFromQValues()** accepterar en inparameter; ett tillstånd *state*. Metoden skall returnera handlingen *action* som get det största q-värdet *Q(state, action)* bland alla giltiga handlingar *actions* som kan utföras i tillståndet *state*. Dock om inga giltiga handlingar finns för tillståndet (t.ex. för ett terminaltillstånd) skall värdet *None* returneras. Även här kan det vara lämpligt att använda superklassens metod *getLegalActions(state)*. Observera att bästa resultat fås om ni slumpar ut en handling om det råkar finnas två eller fler bästa handlingar. Detta kan göras med metoden **random.choice()**. Tänk också på att handlingar som agenten inte har sett innan i ett visst tillstånd har q-värdet 0, dvs om alla andra handlingar har negativa q-värden, kan en handling som inte påträffats innan vara ett optimal handling.

OBS! Se till så att ni endast accessar q-värden genom att anropa **getQValue()** i metoderna **computeValueFromQValues()** och **computeActionFromQValues()**. Detta kommer att hjälpa er när ni löser uppgiften i kapitel 5.1.

Metoden **update()** skall innehålla själva q-learning algoritmen. Den accepterar parametrarna *state*, *action*, *nextState* och *reward* och skall uppdatera ett Q(s,a)-värde. Vi kan dock inte använda nedanstående uppdateringsregel för Q(s,a)-värden eftersom vi inte känner till *T(s,a,s')* eller *R(s,a,s')*, dvs vi har ett okänt MDP problem.

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

Dock kan vi implementera off-policy q-learning algoritmen baserad på sampel-estimat och ett exponentiellt glidande medelvärde enligt nedan:

$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha)[sample]$$

Med $r = R(s,a,s')$ kan detta också skrivas på en rad som:

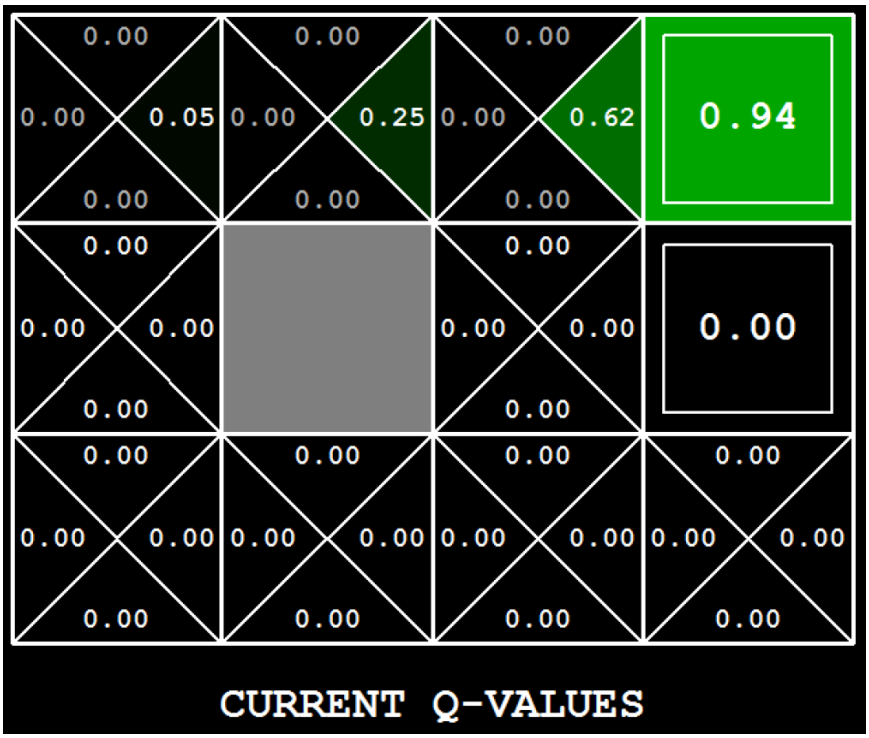
$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha)[R(s, a, s') + \gamma \max_{a'} Q(s', a')]$$

Observera att parametern *gamma* "γ" i ovanstående uppdateringsformel heter *discount* i superklassen **ReinforcementAgent** medans *alpha* "α" faktiskt heter *alpha*.

När ni har implementerat ovanstående metoder kan ni testa inlärningsalgoritmen med följande Python konfiguration:

- gridworld.py -a q -k 5 -m

Men ovanstående konfiguration kommer agenten att lära sig under fem episoder (*-k 5*). Dessutom kommer inlärningen att ske medans ni styr agenten manuellt med piltangenterna på tangentbordet (*-m*). Ni kan se hur agenten lär sig då den lämnar ett tillstånd (uppdateringen sker alltså inte i det nya tillståndet som agenten går till utan *i tillståndet som agenten lämnar*). Om ni vill debugga eran kod genom att styra agenten manuellt genom Gridworld, kan det hjälpa att stänga av *noise* genom att lägga till växeln *--noise 0.0* i ovanstående konfiguration. Om ni manuellt styr agenten norrut och sedan österut till terminaltillståndet längst upp till höger under fyra episoder skall ni se nedanstående bild om ni har implementerat eran q-learning agent korrekt.



När eran agents inlärningsalgoritm fungerar som den skall, kan ni implementera metoden **getAction()**. Metoden accepterar den enda inparametern *state* och skall returnera en handling *action*. Ni skall implementera *epsilon-greedy (ϵ-greedy) action selection* i denna metoden, dvs med sannolikheten ϵ skall metoden returnera en slumpvis handling bland samtliga giltiga handlingar, annars, med sannolikheten $1-\epsilon$, skall metoden returnera den bästa handlingen enligt nuvarande q-värden. Observera att den slumpade handlingen också kan utgöras av den bästa handlingen. För att välja slumpvisa handlingar likformigt kan ni använda metoden **random.choice()**. För att simulera en binär slumpvariabel kan ni använda metoden **utils.flipCoin(p)**, som returnerar *True* med sannolikheten *p* och *False* med sannolikheten *1-p*.

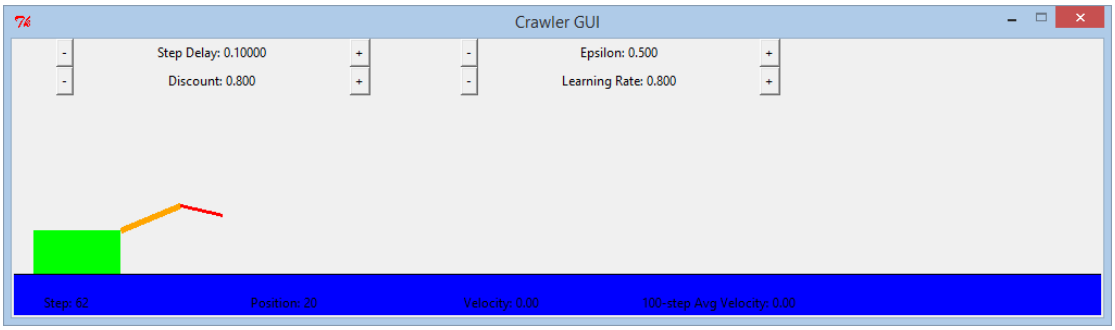
Ni kan testa eran agent med nedanstående Python konfiguration (100 episoder):

- gridworld.py -a q -k 100

Med en korrekt implementerad agent skall era q-värden efterlika era q-värden för value iteration agenten på samma problem. Speciellt för välbesökta vägar.

Dessutom, om eran agent fungerar som den skall, kan ni testa eran q-learning agent på *Crawler* roboten med följande Python konfiguration:

- crawler.py



Om inte detta fungerar så har ni antagligen inte skrivit generell kod för eran q-learning agent (utan skrivit specifik kod för Gridworld problemet). Experimentera med de olika parametrarna för roboten för att se hur de påverkar robotens policy och handlingar.

4.2 BridgeGrid

Testa eran q-learning agent på BridgeGrid problemet med följande Python konfiguration (inlärning under 50 episoder med *noise*=0, *epsilon*=1 och *alpha*=0.9) och observera om agenten lyckas hitta den optimala policyn (value iteration agenten lyckades ju med detta för samma problem) :

- gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1 -l 0.9

Kör samma experiment med *epsilon*=0 och observera om agenten lyckas hitta den optimala policyn:

- gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 0 -l 0.9

Testa att variera ϵ*-greedy* värdet *epsilon* (*-e*) och inlärningshastigheten *alpha* (*-l*). Lyckas eran agent att hitta den optimala policyn med någon kombination av dessa två parametrarna? Fyll i era parametervärden för *epsilon* och *alpha* i laborationsrapporten för laboration 2 (*Rapport_Laboration_2.docx*). Om ni inte tror att någon kombination av de två parametrarna kan hitta den optimala policyn med endast 50 episoder, skriv *GÅR EJ* som värde för de båda parametrarna. Motivera också erat svar, dvs varför hittas den optimala policyn med en viss parameterkombination eller varför går det inte att hitta den optimala policyn med någon parameterkombination?

5 Approximativ Q-Learning och Tillståndsabstraktion

I filen **qlearningAgents.py** finns klassen **PacmanQAgent** definierad. Den ärver samtliga attribut och metoder från klassen **QLearningAgent**, men överrider basklassens metod **getAction()** samt definierar en egen konstruktor **__init__()**.

Klassen **PacmanQAgent** är redan färdigimplementerad, dvs den återanvänder q-learning algoritmen från **QLearningAgent**, fast med andra defaultvärden för *epsilon* (0.05), *gamma* (0.8), *alpha* (0.2) och *numTraining* (0) i konstruktorns parameterlista. Dessa defaultvärden passar bättre för Pacman-problemet.

Testa q-learning agenten för Pacman med nedanstående Python konfiguration:

- pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

Ovanstående konfiguration kör ett antal Pacman spel i två faser. Den första fasen utgörs av en *träningsfas*, där Pacman-agenten kommer att lära sig värdet av olika tillstånd och handlingar. Eftersom det tar ganska lång tid att lära sig bra q-värden även för en liten layout, körs samtliga spel i träningsfasen utan grafik. När samtliga spel i träningsfasen has spelats klart, växlar Pacman-agenten över till *testfasen*. I testfasen kommer Pacman-agentens *self.epsilon* och *self.alpha* parametrar att sättas till *0.0* för att stänga av q-learning och utforskning. Detta så att Pacman-agenten kan utnyttja den inlärda policyn fullt ut. Spel som körs i testfasen kommer att visas grafiskt som default. I ovanstående konfiguration kommer 2010 spel att köras (*-n 2010*) varav de första 2000 av dessa utgörs av spel i träningsfasen (*-x 2000*) och 10 spel tillhör testfasen (dvs de sista 10 spelen kommer att visas grafiskt). Under träningen kommer statistik att presenteras för varje 100 träningsspel. Tänk dock på att *epsilon* har ett värde skiljt från *0* under träningen, dvs Pacman-agenten kan fortfarande spela dåligt efter att en bra policy har hittats (eftersom agenten kommer att utföra slumpvisa utforskande handlingar med sannolikheten *epsilon*). Det borde ta ungefär 1000-1400 spel innan statistiken för 100 träningsspel visar positiva värden för agentens belöning.

Om eran **QLearningAgent** agents kod är korrekt och generellt skriven kommer ovanstående konfiguration att köras felfritt samt Pacman agenten kommer att vinna minst 80% av de tio spelen i testfasen. Om **QLearningAgent** lär sig en bra policy för **gridworld.py** och **crawler.py** men inte för Pacman med ovanstående konfiguration kan detta bero på att **getAction()** och/eller **computeActionFromQValues()** inte hanterar tidigare osedda handlingar på ett korrekt sätt, dvs eftersom osedda handlingar har ett q-värde lika med 0 och om samtliga redan beprövade handlingar har ett negativt värde, kan en osedd handling vara den optimal handlingen. Förlita er inte på **argmax()** metoden i klassen **utilCounter**!

Om ni vill experimentera med inlärningsparametrarna kan ni lägga till växeln *-a* i ovanstpående konfiguration, t.ex. `-a epsilon=0.1,alpha=0.3,gamma=0.7`. Dessa värden kan kommas åt i koden via *self.epsilon*, *self.alpha* och *self.gamma*.

Tänk på att ett MDP-tillstånd utgörs av en exakt brädkonfiguration, där en övergång från ett tillstånd till ett annat, dvs en "ply", består av flera komplexa handlingar. En situation i koden där Pacman-agenten har flyttat till ett annat tillstånd, men inte samtliga spöken är alltså inte ett MDP-tillstånd.

Även om Pacman-agenten spelar bra på *smallGrid* layouten ovan, kommer agenten inte att spela bra på den något större *mediumGrid* layouten:

- pacman.py -p PacmanQAgent -x 2000 -n 2010 -l mediumGrid

Anledningen till varför Pacman-agenten inte vinner på större brädlayouter beror på att varje brädkonfiguration (med Pacmans position, samtliga spökens positioner samt samtliga mat-prickar och tabletter, mm) är ett unikt tillstånd med ett eget q-värde. Pacman-agenten kan alltså inte generalisera genom att lära sig att händelsen "att springa in i ett spöke" inte är bra oavsett brädkonfiguration där detta inträffar. Vi kan återgärda detta med *approximativ q-learning*.

5.1 Approximativ Q-learning

I filen **qlearningAgents.py** finns klassen **ApproximateQAgent** definierad, där ni skall implementera en approximativ q-learning agent, som lär sig en viktad linjär funktion av *features* för att representera samtliga q-värden (där flera tillstånd kan dela på samma *features*). **ApproximateQAgent** ärver från klassen **PacmanQAgent**, men överrider basklassens två metoder **getQValue()** och **update()** samt definierar metoderna **getWeights()**, **final()** och en egen konstruktor **__init__()**.

I konstruktorn **__init__()** skapas en instansvariabel *self.weights* av typ *util.Counter* (en specialiserad *Python dictionary*). Denna instansvariabeln kommer att innehålla den approximativa q-learning algoritmens vikter. Metoden **getWeights()** returnerar denna instansvariabeln.

Approximativ Q-learning använder en *feature*-funktion *f(s,a)* som accepterar ett tillstånd *s* samt en handling *a* som inparametrar och returnerar en vektor med *features*. Ni behöver inte skriva dessa feature-funktioner själva, utan de finns redan implementerade i filen **featureExtractors.py**. Default används *feature*-funktionen **IdentityExtractor** som sätts i konstruktorn. Det enda ni behöver tänka på i koden är att anropa feature-funktionen med nedanstående kod, som returnerar en *feature*-vektor som representeras av en *util.Counter* (en specialiserad *Python dictionary*) där elementen utgörs av paren *{featureNamn, featuerVärde}*, ett för varje *feature*:

```
self.featExtractor.getFeatures(state, action)
```

Metoden **getQValue()** accepterar två inparametrar; ett tillstånd *state* och en handling *action* samt skall returnera det approximativa Q-värdet *Q(state,action)*. Med andra ord skall ni implementera den approximativa Q-funktionen nedan i denna metoden:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

där varje vikt w_i hör ihop med en feature $f_i(s,a)$, och $i=1..n$. I eran kod vill ni implementera vikt-vektorn som en *util.Counter*, där varje element utgörs av paren *{featureNamn, viktVärde}*. **OBS!** Det är viktigt att ni endast accessar q-värden genom att anropa **getQValue()** i basklassen **QLearningAgent** eftersom ni överrider denna metoden här och vill använda eran nya metod **getQValue()** för att beräkna samtliga approximativa q-värden.

Metoden **update()** accepterar fyra inparametrar; ett tillstånd *state*, en handling *action* det efterföljande tillståndet *nextState* samt en belöning *reward* och skall implementera viktuppdateringen i den approximativa q-learning algoritmen nedan:

$$w_i \leftarrow w_i + \alpha \times difference \times f_i(s, a)$$

$$difference = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

Viktuppdateringen skall alltså göras för samtliga vikter och features, från *1* till *n*, där symbolerna i ovanstående formel motsvarar inparametrarna enligt; *s=state*, *a=action*, *s'=nextState* och *r=reward*. Dessutom motsvara α*=alpha* och γ*=gamma*.

Metoden **final()** anropas i slutet av varje spel och kan t.ex. användas för att skriva ut era vikter om ni vill debugga dem.

Ni kan testa eran approximativa q-learning agent med följande Python konfiguration (denna konfigurationen använder *feature*-funktionen **IdentityExtractor**):

- pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

Om ni har implementerat eran kod korrekt, skall eran approximativa q-learning agent fungera på samma sätt som eran **PacmanQAgent** på *smallGrid* layouten.

När eran approximativa q-learning agent fungerar som den skall, kan ni testa agenten på nedanstående Python konfiguration (som använder den bättre *feature*-funktionen **SimpleExtractor** på den större layouten *mediumGrid*):

- pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

Eran approximativa q-learning agent borde vinna nästan varje gång även med endast 50 träningsepisoder.

6 Optimering med den Genetiska Algoritmen

Ett vanligt sätt att optimera vikter på, t.ex. vikterna i ett neuralt nätverk eller vikterna i en viktad linjär evalueringsfunktion till ett spel (som i kapitel 6.5 i laboration 1), är att använda en evolutionär algoritm såsom den genetiska algoritmen. För att få en bättre känsla för hur den genetiska algoritmen fungerar, kommer ni dock att använda algoritmen till att optimera simpla matematiska funktioner istället, där funktionens parametrar utgör "vikterna". Mer generellt kan man säga att den genetiska algoritmen söker efter den bästa lösning på ett problem i en rymd av möjliga lösningar. För en reellvärd funktion av två variabler, dvs $y = f(x_1,x_2)$, utgörs denna sökrymd av den punktmängd som spänns upp av 3-tupeln (y, x_1, x_2) för alla möjliga kombinationer av värden för de två variablerna x_1 och x_2. För varje variabel x_i kommer dessa värden från en viss domän, t.ex. domänen som består av samtliga reella tal, samtliga heltal eller de booleska talen 0 och 1. Denna sökrymd skulle bli aldeles för stor för en generell reellvärd funktion av reellvärda variabler, varför domänen för varje variabel x_i oftast diskretiseras och begränsas. Exempelvist skulle en domän bestående av decimaltal mellan -1.0 och 1.0 med en resolution (upplösning) på 0.1 utgöras av 21 stycken tal från mängden { -1.0, -0.9, ..., 0, ..., 0.9, 1.0 }. Om problemet består av att maximera funktionen $y = f(x_1,x_2)$, där värdena för x_1 och x_2 kommer från domän D_1 respektive D_2, så skulle funktionen $f(x_1,x_2)$ kunna utgöra våran fitnessfunktion, eftersom fitnessfunktionen i den genetiska algoritmen talar om hurpass "bra" en individ är på att lösa problemet, dvs ju högre värde desto bättre. Om problemet består av att minimera funktionen $y = f(x_1,x_2)$, så kan vi helt enkelt använda $-f(x_1,x_2)$ som våran fitnessfunktion istället.

När man använder den genetiska algoritmen för att lösa ett specifikt problem måste man 1) representera problemet på något lämpligt sätt i en bitsträng (kromosom) samt 2) skapa en fitnessfunktion som indikerar hur "bra" en lösning (individ) är. Utöver detta behöver man välja värden på den genetiska algoritmens parametrar, t.ex. populationsstorlek, selektionsmetod, crossoversannolikhet, mutationssannolikhet, termineringskreterie, mm. För hamburgerproblemet, som vi gick igenom under föreläsningen om den genetiska algoritmen, var detta trivialt, där problemet representerades med tre booleska variabler (tre bitar) i kromosomen (bitsträngen) och fitnessfunktionen beräknade heltalet som motsvarade bitsträngens binärkod. För andra problem behöver man vara mer kreativ med avseende på problemrepresentation samt fitnessfunktion. Det är också viktigt hur man ställer in den genetiska algoritmens parametrar för ett visst problem. Ni kommer dock att få några tumregler för dessa parameterar som ni kan utgå ifrån (endast experimenterande kan ge optimala värden för dessa parametrar för komplexa problem).

Den viktade linjära evalueringsfunktionen som användes i kapitel 6.5 i laboration 1, utgjorde fitnessfunktionen för det problemet, där vikterna i evalueringsfunktionen utgjorde variablerna x_i. För att optimera vikterna för Pacman-problemet skulle man helt enkelt kunna skapa en population med ett antal individer, där varje individ representerade variablerna x_i som bitar i en bitsträng på något sätt, använda den viktade linjära evalueringsfunktionen som fitnessfunktion samt välja parametrarna för den genetiska algoritmen på lämpligt sätt. Därefter är det bara att starta körningen av den genetiska algoritmen, som itererar igenom ett antal generationer, varpå den bästa lösningen på problemet utgörs av den bästa individen (bitsträngen) i den sista generationen.

Varje gång vi beräknar fitnessen för en individ startar vi helt enkelt ett nytt pacman-spel, använder individens variabler som vikterna i den viktade linjära evalueringsfunktionen och spelar spelet från start till slut. Antalet poäng (score) skulle då utgöra fitnessvärdet på individen. För funktionsoptimeringsproblemet fungerar detta på exakt samma sätt, där funktionen utgör fitnessfunktionen (för maximeringsproblem) och variablerna utgör vikterna. Det enda som skiljer sig från pacman-problemet är att vi inte har några explicit valda features i vårat funktionsoptimeringsproblem samt att vi helt enkelt beräknar funktionsvärdet istället för att "spela igenom något spel". För båda problemen måste vi dock representera variablerna som bitar i en bitsträng och avkoda dessa på något sätt. I denna laborationen kan vi göra detta genom att använda *standard binär kodning* (se nedan).

Ni skall först komplettera ett enkelt framework (i filen **genetic_algoritm.py**) för problemlösning med den genetiska algoritmen. Kromosomerna representeras av en bitsträng av godtycklig längd (dock med fast längd för ett specifikt problem). Standardvarianter på *roulette-wheel selection*, *tournament selection*, *crossover* och *mutation* skall användas. Det enda som behöver vara problemspecifikt är representationen (hur problemet representeras i form av en kromosom med ett antal gener) och fitness-funktionen (hur fitness-värdet för en individ beräknas). För att verifiera att erat GA-ramverk fungerar, skall ni maximera fyra funktioner:

$$f_1(x) = -x^2 - 4x - 2 \tag{1}$$

$$f_2(x) = -x^4 + x^3 + 4x^2 - 2x - 5 \tag{2}$$

$$f_3(x_1, x_2) = \frac{1}{1 + (x_1 + x_2 + 1)^2 (19 - 14x_1 + 3{x_1}^2 - 14x_2 + 6x_1x_2 + 3{x_2}^2)} \times \frac{1}{30 + (2x_1 - 3x_2)^2 (18 - 32x_1 + 12{x_1}^2 + 48x_2 - 36x_1x_2 + 27{x_2}^2)} \tag{3}$$

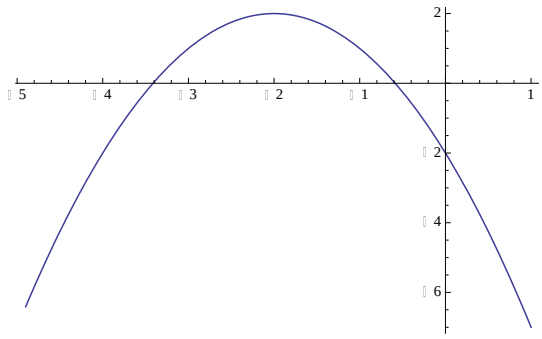
$$f_4(x_1, x_2) = 0.5 - \frac{\left(\sin\sqrt{x_1^2 + x_2^2}\right)^2 - 0.5}{\left(1 + 0.001(x_1^2 + x_2^2)\right)^2} \tag{4}$$

Funktionen $f_1(x)$ har ett globalt maximum i punkten $\{x, f_1(x)\} = \{-2,2\}$, d.v.s. $f_1(x) = 2$ då $x = -2$ (se Figur 6.1).

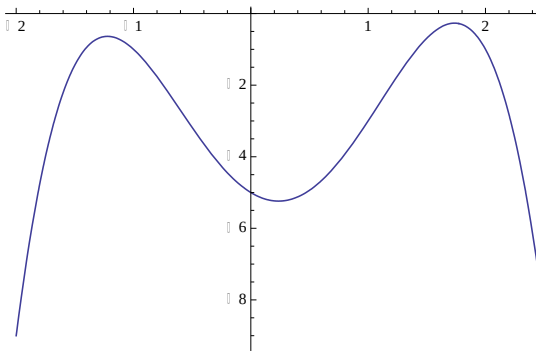
Funktionen $f_2(x)$ har ett globalt maximum i punkten $\{x, f_2(x)\} = \{1.73611, -0.267803\}$, d.v.s. $f_2(x) = -0.267803$ då $x = 1.73611$ (se Figur 6.2).

Funktionen $f_3(x_1, x_2)$ har ett globalt maximum i punkten $\{x_1, x_2, f_3(x_1, x_2)\} = \{0, -1.0, 0.333\}$, d.v.s. $f_3(x_1, x_2) = 0.333$ då $x_1 = 0\ och\ x_2 = -1.0$ (se Figur 6.3).

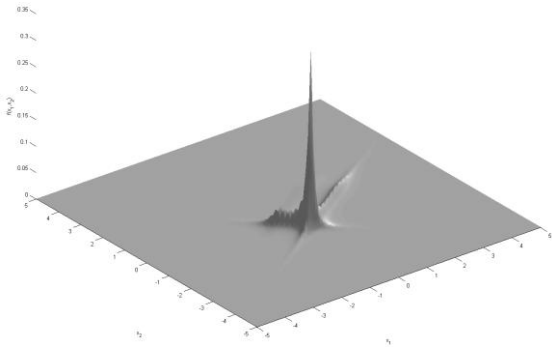
Funktionen $f_4(x_1, x_2)$ har ett globalt maximum i punkten $\{x_1, x_2, f_4(x_1, x_2)\} = \{0, 0, 1.0\}$, d.v.s. $f_4(x_1, x_2) = 1.0$ då $x_1 = x_2 = 0$ (se Figur 6.4).



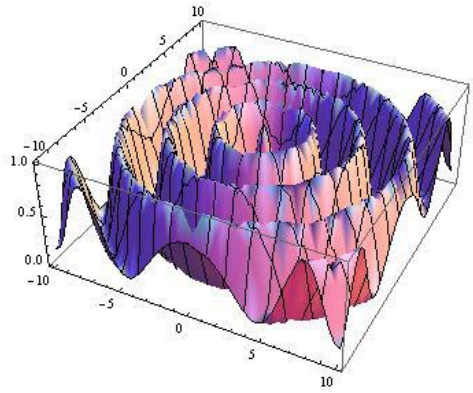
Figur 6.1 $f_1(x)$



Figur 6.2 $f_2(x)$



Figur 6.3 $f_3(x_1, x_2)$



Figur 6.4 $f_4(x_1, x_2)$

De fyra funktionerna (fitnessfunktionerna) definieras längst upp i filen **function_optimization.py**. En klass finns definierad för varje fitnessfunktion, där klassen även innehåller information om antalet variabler som ingår i fitnessfunktionen och varje variabels domän. Övrigt kod i filen **function_optimization.py**, skapar ett GUI som ni kan använda för att ställa in de olika parmetrarna för den genetiska algoritmen samt för att exekvera den genetiska algoritmen på de fyra matematiska funktionerna. I metoden **config_changed()** läses samtliga parameterinställningar från GUI:t av och en instans av den genetiska algoritmen skapas, där parameterinställningarna skickas in som argument till klassens konstruktor **GA(...)**. Detta är det enda ni behöver känna till i denna filen för att kunna genomföra laborationen.

Filen **genetic_algorithm.py** innehåller själva ramverket för den genetiska algoritmen. För att applicera den genetiska algoritmen på ett visst problem, skapar man alltså en instans av klassen **GA**, där samtliga parameterinställningar skickas in som argument till konstruktorn. Därefter kan man anropa metoden **Run()** för att köra algoritmen fram tills termineringskriteriet (max antal generationer) har uppfyllts eller så kan man aropa metoden **Step()** för att stega igenom algoritmen en generation i taget. Den bästa individen (bitsträngen) med tillhörande fitnessvärde erhålls därefter från den instanserade **GA** klassen via uttrycket **ga.population[ga.bestIndividualIndex]** respektive **ga.maximumFitness**. Förrutom konstruktorn **__init__()** och de två metoderna **Step()** och **Run()** så innehåller filen metoderna **CalculateFitness()**, **InitializePopulation()**, **DecodeChromosome()**, **RouletteWheelSelect()**, **TournamentSelect()**, **Cross()**, **Mutate()** och **InsertBestIndividual()**.

Konstruktorn **__init__()** accepterar parametrarna enligt nedan:

- *populationSize* : antalet individer (bitsträngar) som ingår i populationen
- *numberOfGenes*: totala antalet bitar i en bitsträng (individ)
- *crossoverProbability*: sannolikheten för att två förälderindivider skall korsas
- *mutationProbability*: sannolikheten för att respektive bit i en bitsträng skall "flippas"
- *selectionMethod*: 0 => Tournament Selection, annars => Roulette-Wheel Selection
- *tournamentSelectionParameter*: turneringens sannolikhetsvärde (se P_{tour} nedan)
- *tournamentSize*: antalet individer i varje turnering
- *numberOfVariables*: antalet variabler som representeras i en bitsträng
- *variableRange*: intervallet för en variabels domän (som en *numpy* array)
  - Exempel med 1 variabel: `np.array([[-2.0,2.5]])`
  - Exempel med 2 variabler: `np.array([[-1,1],[-2,0]])`
- *numberOfGenerations*: max antal generationer (termineringskriteriet i detta fallet)
- *useElitism*: booleskt värde som indikerar ifall *Elitism* (se nedan) används eller inte
- *numberOfBestIndividualCopies*: antalet *Elite* kopior om *Elitism* (se nedan) används
- *fitnessFunction*: en referens till en funktion som utgör fitnessfunktionen

Konstruktorn **__init__()** sparar undan en referens till samtliga inparametrar och beräknar sedan den initiala populationens fitness genom att anropa **CalculateFitness()**. Konstruktorn är redan färdigimplementerad.

Den parameterlösa metoden **CalculateFitness()** är också färdigimplementerad. Metoden börjar med att avkoda varje individ, dvs omvandlar en individs bitsträng till ett eller flera variabler, genom att anropa metoden **DecodeChromosome()**. Därefter evalueras varje individ genom att anropa metoden **EvaluateIndividual()** med variablerna som inparametrar. Slutligen beräkns varje individs fitness samt populationes totala-, medel-, min- och max fitness. Indexet till den bästa individen i populationen sparas också undan.

Metoden **InitializePopulation(populationSize, numberOfGenes)** är också redan färdigimplementerad och tar populationsstorleken samt antalet gener i en kromosom (antalet bitar i en bitsträng) som inparametrar och returnerar en tvådimensionell *numpy array* bestående av **populationSize** (antal rader) och **numberOfGenes** (antal kolumner), dvs varje rad utgör en bitsträng (individ) med **numberOfGenes** antal bitar. Dessutom så är värdet (0 eller 1) för varje bit i varje bitsträng helt slumpmässigt vald. Den returnerade tvådimensionella *numpy array*en utgör alltså populationen.

*Numpy array*er är speciella arrayer som tillhör *scipy stacken*, vilket ni kommer att använda flitigt under dataanalysdelen i kursen. Därför introduceras de försiktigt här i denna laborationen så att ni kan bekanta er med dem redan nu. Skumma igenom den *Numpy tutorial* (fram till *Functions and Methods Overview*) som finns på webbsidan *http://wiki.scipy.org/Tentative_NumPy_Tutorial* (bry er inte om den linjära algebran och de komplexa talen). Skumma även igenom metoderna *arange(), array(), copy(), empty(), ones(), zeros(), reshape(), resize(), max(), min(), rand(), random_integers()* och *size()* på webbsidan *http://wiki.scipy.org/Numpy_Example_List*.

Metoden **Run()** itererar igenom *numerOfGenerations* antal generationer genom att anropa metoden **Step()** *numerOfGenerations* antal gånger.

Metoden **Step()** innehåller själva stommen för den genetiska algoritmen. Först skapas en temporär *Numpy array* för nästa generation. Därefter går metoden in i en loop som upprepas tills den temporära arrayen har fyllts med lika många individer som finns i nuvarande generation, dvs *populationSize* antal individer. Inuti loopen används först en selektionsstrategi (*Tournament Selection* eller *Roulette-Wheel Selection*) för att välja ut två föräldrarindivider. Därefter korsas de båda föräldrarindividerna (med en viss sannolikhet) för att skapa två barnindivider, vilka därefter muteras (med en viss sannolikhet muteras varje bit i varje barnindivids bitsträng). Slutligen läggs barnindividerna till i den temporära *Numpy array*en. När den temporära arrayen har fyllts, avbryts loopen, varpå den bästa individen i nuvarande generation kopieras oförändrad in i den temporära arrayen (om *Elitism* används). Innan metoden returnerar, ersätts nuvarande generations array (den nuvarande populationen) med den temporära arrayen (nästa generations population) samt fitnessen beräknas för den nya generationens population. Ni skall komplettera de halvfärdiga metoderna som anropas i huvudloopen, dvs **TournamentSelect(), RouletteWheelSelect()**, **Cross()** och **Mutate()** samt metoden **DecodeChromosome()**. Dock, innan dessa metodena beskrivs följer lite teori nedan.

Först måste en bitsträng kunna avkodas till en eller flera variabler. Ni kommer att implementera detta med hjälp av *standard binär kodning*.

6.1 Standard binär kodning

I standard binär kodning kan varje gen i en kromosom endast anta värdet 0 eller 1 (dvs en binär sträng). Generna g_1, .., g_k i bitsträngen avkodas enligt

$$x = l + \frac{u-l}{1-2^{-k}}(2^{-1}g_1 + \cdots + 2^{-k}g_k)$$

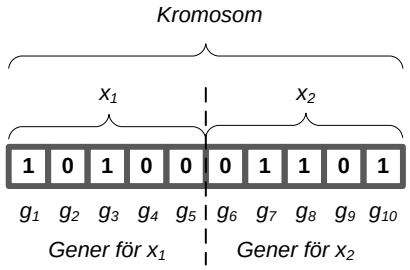
för att erhålla ett värde på variabeln *x* inom intervallet [*l*,*u*]. Det lägsta inkrementet (precisionen) för variabeln *x* blir därför

$$\frac{u-l}{1-2^{-k}}2^{-k}$$

Ju fler bitar i bitsträngen (kromosomen), desto högre precision för variabeln *x*. Till exempel, om vi representerar de två variablerna x_1 samt x_2, vardera med *k=5* bitars precision, inom intervallet [-10,10] (dvs *l=-10*, *u=10*), i en kromosom, får vi minsta inkrementet (precisionen) för respektive variabel enligt

$$\frac{u-l}{1-2^{-k}}2^{-k} = \frac{10-(-10)}{1-2^{-5}} \times 2^{-5} \approx 0.645161$$

Om vi har en instans av kromosomen enligt nedanstående figur



kan vi avkoda variablerna x_1 samt x_2 enligt

$$x_1 = l + \frac{u-l}{1-2^{-k}}(2^{-1}g_1 + \cdots + 2^{-k}g_5) =>$$
$$x_1 = -10 + \frac{10-(-10)}{1-2^{-5}}(2^{-1} \times 1 + 2^{-2} \times 0 + 2^{-3} \times 1 + 2^{-4} \times 0 + 2^{-5} \times 0) =>$$
$$x_1 = -10 + \frac{20}{1-2^{-5}}(2^{-1} + 2^{-3}) = \frac{90}{31} \approx 2.90323$$

$$x_2 = l + \frac{u-l}{1-2^{-k}}(2^{-1}g_6 + \cdots + 2^{-k}g_{10}) =>$$
$$x_2 = -10 + \frac{10-(-10)}{1-2^{-5}}(2^{-1} \times 0 + 2^{-2} \times 1 + 2^{-3} \times 1 + 2^{-4} \times 0 + 2^{-5} \times 1) =>$$
$$x_2 = -10 + \frac{20}{1-2^{-5}}(2^{-2} + 2^{-3} + 2^{-5}) = -\frac{50}{31} \approx -1.6129$$

Skulle vi vilja ha heltal istället kan vi t.ex. använda oss av trunkering. Exempelvis får vi x_1=2 om vi trunkerar värdet 2.90323. På så sätt kan ni alltså representera reella variabler (vikter), med en viss precision, inom ett visst intervall, som en bitsträng (kromosom).

6.2 "Pseudokod" för en standardimplementering av GA

Nedan ges pseudokod för en standardvariant av den genetiska algoritmen.

1. Initiera populationen (motsvaras av metoden **InitializePopulation()** i koden)
   Initiera populationen genom att slumpmässigt generera N binära strängar (kromosomer) c_i, i=1,…,N med längd $m=kn$, där k är antal bitar per variabel, och n är antalet variabler.
2. Evaluera individerna (motsvaras av metoden **CalculateFitness()** i koden):
   a. Avkoda kromosom c_i för att erhålla motsvarande variabler x_{ij}, j=1,…,n. Detta motsvaras av metoden **DecodeChromosome()** i koden.
   b. Evaluera den objektiva funktionen f genom att använda variabelvärdena som erhölls i 2.a, och tilldela "fitness"-värdet $F_i=f(x_{i1}, x_{i2}, ..., x_{in})$. Detta motsvaras av anropet till **EvaluateIndividual()** i metoden **CalculateFitness()** i koden. **EvaluateIndividual()** är fitnessfunktionen.
   c. Återupprepa steg 2.a-2.b tills hela populationen har evaluerats.
   d. Sätt $i_{best} = 1$, $F_{best} = F_1$. Loopa sedan igenom samtliga individer; om $F_i > F_{best}$, sätt $F_{best} = F_i$ och $i_{best} = i$ (steg 2.d behövs för "elitism" - se kapitel 6.6).
3. Skapa nästa generation (motsvaras av **Step()** metoden i koden):
   a. Skapa en exakt kopia av den bästa kromosomen c_{ibest} (för "elitism").
   b. Selektera två individer, i_1 och i_2, från den evaluerade populationen, genom att använda en lämplig selektionsmetod (t.ex. "tournament selection" - se kapitel 6.3 eller "roulette-wheel selection" - se kapitel 6.4). Dessa två selektionsmetoder motsvaras av metoderna **TournamentSelect()** och **RouletteWheelSelect()** i koden,
   c. Skapa två barn-kromosomer genom att korsa (t.ex. "single-point crossover"), med sannolikhet p_c, de två förälderkromosomerna c_1 och c_2. Med sannolikhet $1-p_c$, kopieras alltså de två förälderkromosomerna utan någon modifiering (alltså ingen "crossover"). P_c brukar väljas någonstans kring 0.8. Detta motsvaras av metoden **Cross()** i koden.
   d. Mutera de två barn-kromosomerna, dvs för varje gen (i varje barn-kromosom), mutera genen med sannolikhet p_{mut} (se kapitel 6.5). Detta motsvaras av metoden **Mutate()** i koden.
   e. Återupprepa steg 3.b-3.d tills ytterligare N-1 individer har genererats. Ersätt sedan de N gamla individerna med de N nygenererade individerna.
   f. Kopiera in N_{best} kopior av den bästa kromosomen c_{ibest} i den nya generationen (för "elitism"). Detta motsvaras av metoden **InsertBestIndividual()** i koden.
4. Upprepa (motsvaras av **Run()** metoden i koden)
   Återvänd till steg 2, om inte terminations-kriteriet har uppfyllts (t.ex. tills max antal generationer har uppnåtts eller t.ex. tills den exakta lösningen har erhållits om den är känd).

6.3 Tournament Selection

En simpel variant av *tournament selection* består av att välja två individer, slummässigt (dvs med samma sannolikhet för alla individer, med återläggning) från populationen, varpå den bästa individen av de två, dvs individen med högst "fitness", väljs med sannolikhet P_{tour}. Detta innebär alltså att den sämre individen väljs med sannolikhet $1-P_{tour}$. Parametern P_{tour} kallas för "tournament selection parameter", och sätts vanligtvist till ett värde kring 0.7-0.8. Rent praktiskt, implementeras tournament selection genom att slumpa ett tal r inom intervallet [0,1], varpå den bästa individen av de två väljs om $r < P_{tour}$. Annars väljs den sämre av de två individerna.

Tournament selection kan generaliseras till fallet med fler än två individer. I det generella fallet, väljs j individer (slummässigt med återläggning) från populationen. Därefter selekteras den bästa individen (vinner turneringen) med sannolikhet P_{tour}. Om denna individen inte selekteras, återupprepas proceduren för de kvarvarande $j-1$ individerna, återigen med en sannolikhet P_{tour} för att selektera den bästa individen. Parametern j kallas för "the tournament size". En större tournament size leder till större konkurrens bland individerna. Ni får testa er fram med olika värden på "tournament size", t.ex. 2-4.

6.4 Roulette-Wheel Selection

När man använder *roulette-wheel selection* är sannolikheten för att en individ skall väljas till mating poolen proportionell mot dess fitness relativt populationens totala fitness. Man summerar helt enkelt ihop samtliga individers fitness F_i för att få populationens totala fitness F_{TOT} och delar sedan varje individs fitness med denna summa för att erhålla en individs relativa fitness F_i/F_{TOT}. Man kan likna selektionsprocessen med att snurra på ett roulettehjul, där individens relativa fitness utgör en cirkelsektor i det cirkelfortmade roulettehjulet. Samtliga cirkelsektorer (individers fitness) fyller hela roulettehjulet, där storleken för en individs cirkelsektor är proportionell mot individens relativa fitness. Sannolikheten för att en individ skall väljas till mating poolen är alltså större för en individ med högre fitnessvärde.

Eftersom den relativa fitnessen inte kan beräknas på ovanstående sätt om negativa fitnessvärden tillåts, brukar man oftast beräkna den normaliserade fitnessen för varje individ först F_{Ni}. Detta kan t.ex. göras genom att subtrahera populationens min fitness från en individs fitness och dela med skillnaden mellan populationens max- och min fitness, dvs $F_{Ni} = (F_i-F_{MIN})/(F_{MAX}-F_{MIN})$, för att erhålla en fitnessskala på [0, 1]. Om man summerar ihop dessa normaliserade fitnessvärden får man istället populationens totala normaliserade fitness som F_{NTot} istället. Nu fås den relativa fitnessen som F_{Ni}/F_{NTot} istället. Rent praktiskt kan man slumpa ett värde $r \in [0,1]$ och loopa igenom samtliga individer där deras relativa fitness summeras i varje iteration genom loopen tills summan blir minst lika stor som r. Den individ vars index loopräknaren stannde på blir då selekterad.

6.5 Mutation

Parametern för mutationssannolikheten, p_{mut}, sätts vanligtvist till c/m, där c är en konstant och m är längden på bitsträngen (kromosomen). De bästa resultaten erhålls oftast då $c=1$, dvs $p_{mut} = 1/m$. En gen i en binär sträng kan muteras genom "bit flipping", dvs om genen har värdet 0, får den värdet 1. På motsvarande sätt får en gen med värdet 1, värdet 0 efter att den har muterats.

6.6 Elitism

När man använder "elitism", kopieras den bästa individen i varje generation till nästa generation utan någon modifiering (dvs utan någon "crossover" eller mutation). Då elitism används är det alltså denna individen (kromosomen) som väljs som bästa lösning på problemet i den sista generationen. En variant på elitism är att kopiera mer än 1 kopia av den bästa individen till nästa generation.

6.7 Kod

Ni skall ni komplettera de halvfärdiga metoderna **TournamentSelect(), RouletteWheelSelect()**, **Cross(), Mutate()** och **DecodeChromosome()** så att ni kan beräkna ett korrekt maxima för de fyra mattematiska funktikonerna med hjälp av ramverket för den genetiska algoritmen.

Metoden **DecodeChromosome(chromosome, numerOfVariables, variableRange)** accepterar en kromosom **chromosome** (dvs en individ som utgörs av en *Numpy array*), antalet variabler som finns representerade i en bitsträng **numerOfVariables** och intervallet för respektive variabel **variableRange** som inparametrar och skall returnera en en-dimensionell *Numpy array* av variabelvärden. Avkoda bitsträngarna enligt beskrivningen i kapitlet om *standard binär kodning* ovan. För att t.ex. komma åt den undre gränsen för den första variabels intervall kan följande syntax användas:

```
variableRange[0][0].
```

Metoden **RouletteWheelSelect(normalizedFitness)** accepterar en endimensionell array av normaliserade fitnessvärden **normalizedFitness** som inparameter och skall returnera indexet för den individ som selekterades från populationen.

TournamentSelect(fitness, **tournamentSelectionParameter**, **tournamentSize)** accepterar en endimensionell array av fitnessvärden **fitness**, turneringens sannolikhetsparameter P_{tour} **tournamentSelectionParameter** samt turneringens storlek **tournamentSize** som inparameter och skall returnera indexet för den individ som selekterades från populationen.

Metoden **Cross(chromosome1**, **chromosome2**, **crossoverProbability)** accepterar två förälderkromosomer **chromosome1** och **chromosome1** (dvs individer som utgörs av *Numpy array*er) och sannolikheten för att de skall korsas P_c **crossoverProbability** som inparameter. Funktionen skall **inte** returnera någonting, utan om föräldrarkromosomerna korsas med varandra, så skall föräldrarkromosomernas *Numpy array*er **chromosome1** och **chromosome1** modifieras "*in-place*"!

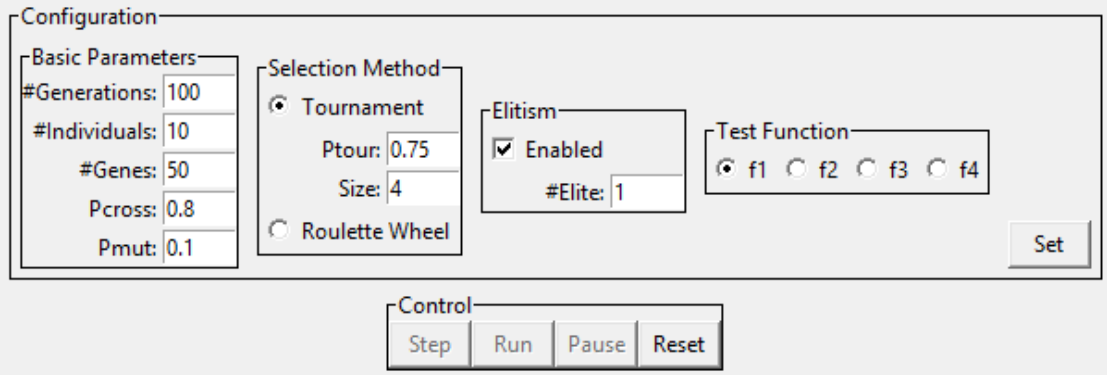
Mutate(chromosome, **mutationProbability)** accepterar en kromosom **chromosome** (dvs en individ som utgörs av en *Numpy array*) och sannolikheten för en bit skall muteras P_m **mutationProbability** som inparameter. Funktionen skall **inte** returnera någonting, utan om en bit muteras, så skall kromosomens *Numpy array* **chromosome** modifieras "*in-place*"!

För att bli godkända på denna uppgiften måste eran kod kunna hitta maximat för de fyra mattematiska funktionerna. Fyll också i era parametervärden för den genetiska algoritmen som ni anser vara optimala för att lösa de fyra mattematiska funktionerna i laborationsrapporten för laboration 2 (*Rapport_Laboration_2.docx*).

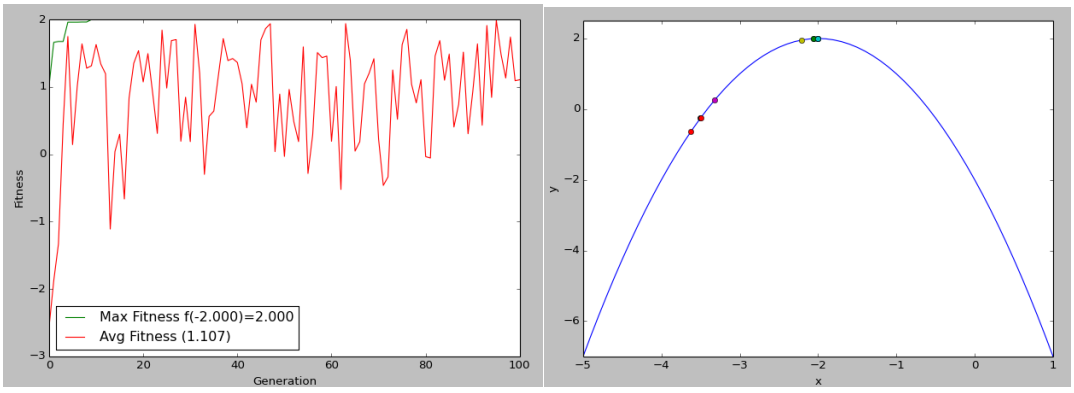
Ni kan använda GUI:t för att visualisera vad som händer när ni ändrar på de olika parametervärdena. Använd nedanstående Python konfiguration för att starta GUI:t ...

function_optimization.py

... och ställ in samtliga parametrar i GUI:t. Klicka på **Set** knappen varje gång ni ändrar på parameterinställningarna. Klicka sedan på **Step** eller **Run** knappen för att köra den genetiska algoritmen.



I den vänstra grafen ser ni hur max- respektive medelfitnessen ändras från generation till generation. I den högra grafen ser ni funktionen som ni vill maximera och samtliga individer som punkter i denna sökrymden.



Om ni inte vill använda GUI:t kan ni helt enkelt skapa en instans av **GA** klassen med samtliga inparametrar instanserade i en annan fil, anropa **ga.Run()** och skriva ut **ga.maximumFitness** när körningen är klar. Ni hittar fitnessfunktionerna för varje mattematisk funktion längst upp i filen **function_optimization.py**.

7 Övrig information

7.1 Handledning

Handledning sker vid tre tillfällen enligt de tider som finns i schemat. Tänk på att komma till handledningen med väl förberedda frågor. Bokningslistor för handledningen finns i PingPong under Aktivitet/Innehåll/Handledning/Lab 2. Där hittas Doodle länkar för varje handledningstillfälle (OBS! En Doodle länk för bokning av ett visst handledningstillfälle blir först synlig i PingPong dagen efter det förra handledningstillfället).

För varje handledningstillfälle (Handledning 1, 2 samt 3), boka endast **ett** handledningspass per grupp.

Med reservation för ändringar (kolla alltid schemat), så gäller för närvarande nedanstående datum (och rum) för varje handledningstillfälle:

- Handledning 1 sker onsdagen den 2 december i rum L433.
- Handledning 2 sker fredagen den 4 december i rum L433.
- Handledning 3 sker torsdagen den 17 december i rum L433.

7.2 Inlämning

Laborationen lämnas in via PingPong, under Aktivitet/Inlämning Lab 2, där samtliga kodfiler samt laborationsrapporten lämnas in som **en** arkivfil (7z, rar, tar eller zip). Följande filer skall finnas med i arkivfilen: **valueIterationAgents.py**, **qlearningAgents.py**, **genetic_algorithm.py** samt **Rapport_Laboration_2.docx**.

Inlämning och examination av laborationen sker vid tre tillfällen enligt schemat (OBS! En länk för ett visst inlämningstillfälle blir först synlig i PingPong dagen efter det förra inlämningstillfället). Samtliga tre tillfällen får utnyttjas. Om ni blir underkända på någon uppgift efter första examinationstillfället skall denna vara åtgärdad innan andra examinationstillfället. Om laborationen fortfarande är underkänd efter andra examinationstillfället, så ges en modifierad version av laborationen ut som skall lämnas in senast efter sommaren 2016. Nedanstående datum gäller för varje examinationstillfälle:

- Deadline för inlämningstillfälle 1 är söndagen den 13 december 23:59.
- Deadline för inlämningstillfälle 2 är söndagen den 17 januari 23:59.
- Deadline för inlämningstillfälle 3 är söndagen den 28 augusti 23:59.

7.3 Betyg

Endast betygen Underkänd eller Godkänd förekommer på laborationen.