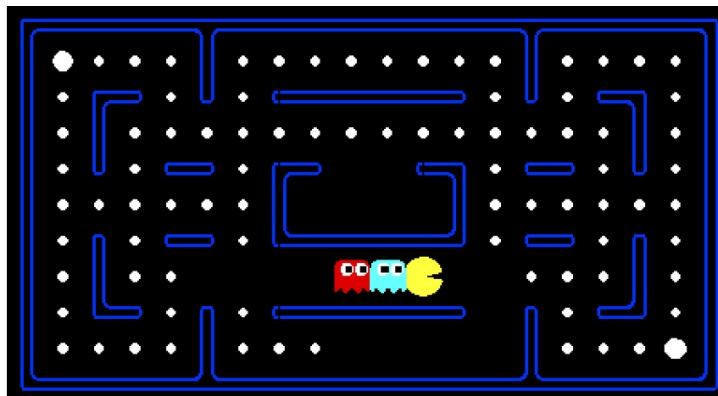
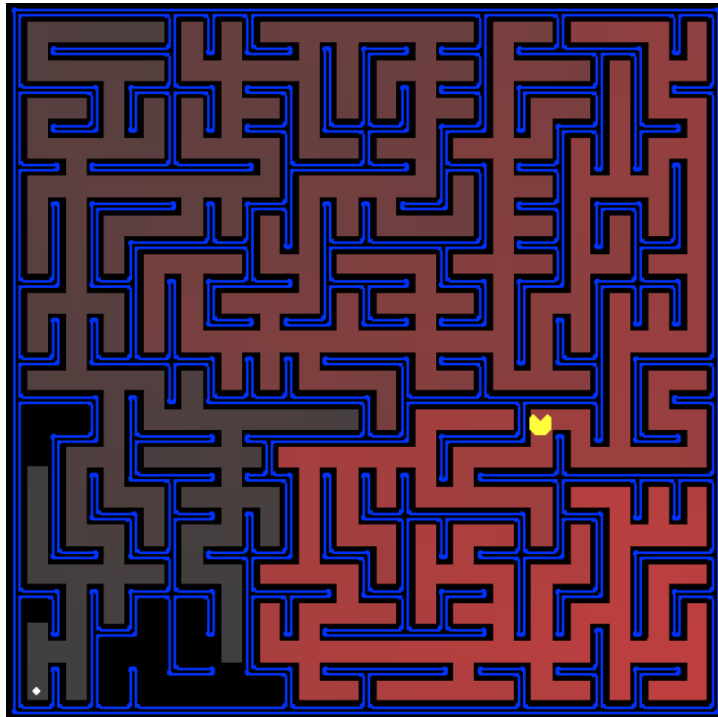


Laboration 1

Sökning i Pacman



Innehåll

1	Inledning.....	- 1 -
2	Installation och Introduktion.....	- 1 -
3	Uppgift 1 - Sökalgoritmer (singelagentmiljöer).....	- 12 -
3.1	Grafsökning (GraphSearch).....	- 15 -
3.2	Djupet-Först Sökning (Depth-First Search).....	- 16 -
3.3	Bredden-Först Sökning (Bredth-First Search).....	- 16 -
3.4	Uniform Kostnad Sökning (Uniform Cost Search).....	- 17 -
3.5	A* Sökning (A* Search).....	- 17 -
4	Uppgift 2 - Problemrepresentation.....	- 18 -
4.1	CornersProblem.....	- 19 -
5	Uppgift 3 - Design av Heuristiska Funktioner.....	- 20 -
5.1	cornersHeuristic.....	- 21 -
6	Uppgift 4 - Sökalgoritmer (multiagentmiljöer).....	- 22 -
6.1	Reflexagent.....	- 23 -
6.2	Minimax.....	- 24 -
6.3	AlphaBetaAgent.....	- 26 -
6.4	ExpectimaxAgent.....	- 27 -
6.5	betterEvaluationFunction.....	- 28 -
7	Övrig information.....	- 29 -
7.1	Handledning.....	- 29 -
7.2	Inlämning.....	- 29 -
7.3	Betyg.....	- 29 -

1 Inledning

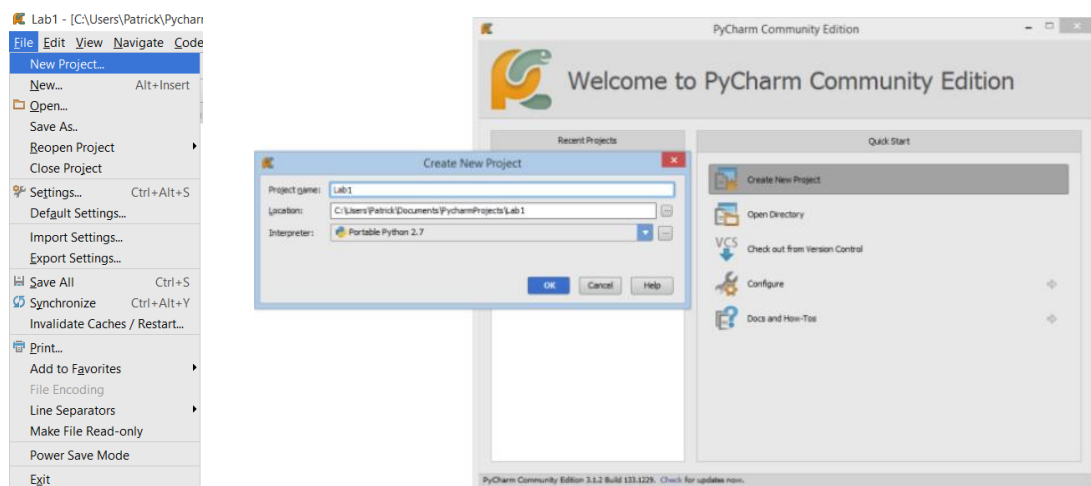
Denna laborationen behandlar *problemrepresentation* och *sökning*. Inledningsvis undersöks sökning i *fullt observerbara, deterministiska, diskreta, statiska singel-agent*problem där sökalgoritmerna **DFS**, **BFS**, **UCS** samt **A*** kommer att implementeras för en problemlösande agent som använder den generella **grafsökningsalgoritmen**. Mer specifikt skall planer tas fram som hjälper Pacman att lösa olika spökfria problem.

Därefter undersöks sökning i *fullt observerbara, deterministiska, diskreta, statiska multi-agent*problem där sökalgoritmerna **minimax** (med och utan **alpha-beta pruning**) och **expectimax** kommer att implementeras för en problemlösande agent. Mer specifikt skall planer tas fram som hjälper Pacman att lösa olika problem som inkluderar spöken.

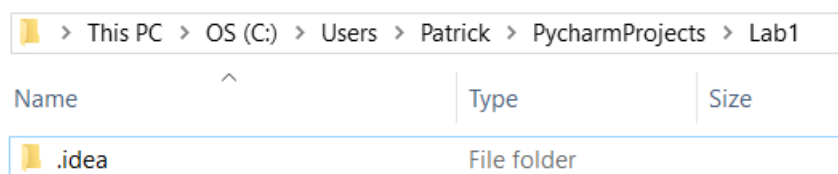
Sist undersöks sökning i *fullt observerbara, deterministiska, diskreta, statiska multi-agent*problem där sökalgoritmerna minimax (med alpha-beta pruning) och expectimax samt en **evalueringsfunktion** kommer att implementeras för en problemlösande agent. Mer specifikt skall planer tas fram som hjälper Pacman att lösa olika *genereliserbara problem* som inkluderar spöken.

2 Installation och Introduktion

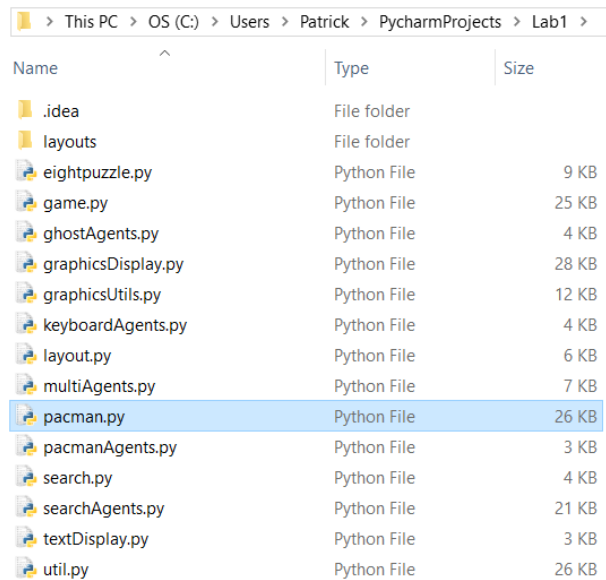
Starta PyCharm och skapa ett nytt projekt (t.ex. med namn "Lab1").



Ladda ner filen Lab1_filer.zip från PingPong och packa upp den i projektfoldern som du skapade ovan.



Filen `pacman.py` skall nu finnas direkt under projektfoldern (i samma folder som katalogen `".idea"` i nedanstående bild och inte i foldern `Lab1_files`).



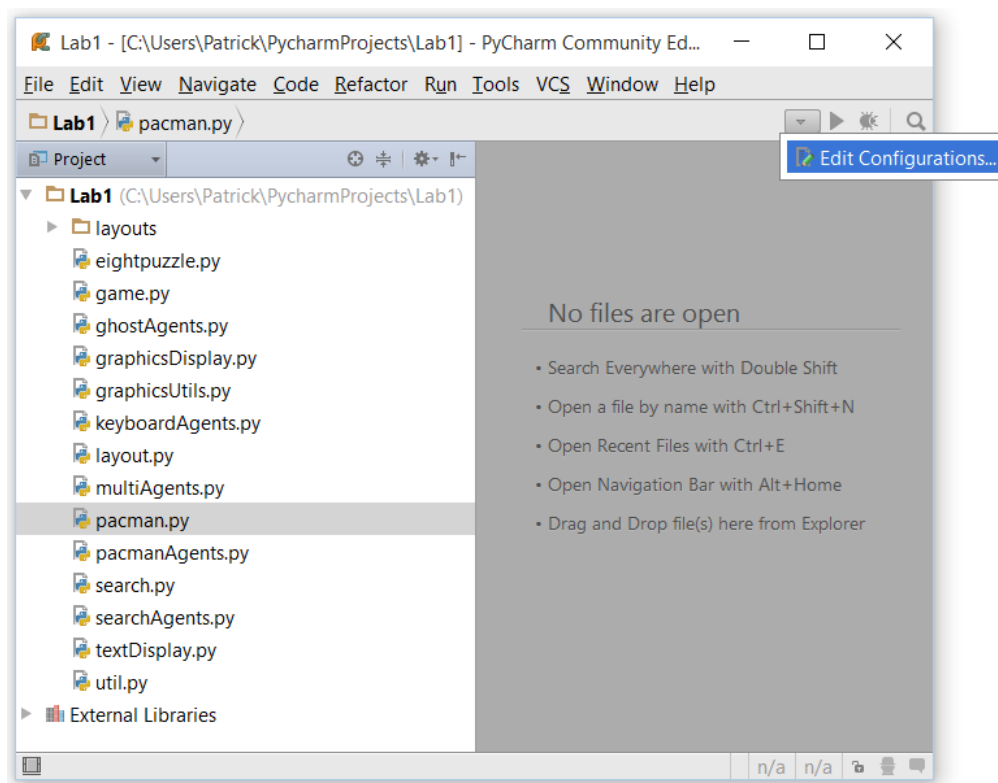
Name	Type	Size
.idea	File folder	
layouts	File folder	
eightpuzzle.py	Python File	9 KB
game.py	Python File	25 KB
ghostAgents.py	Python File	4 KB
graphicsDisplay.py	Python File	28 KB
graphicsUtils.py	Python File	12 KB
keyboardAgents.py	Python File	4 KB
layout.py	Python File	6 KB
multiAgents.py	Python File	7 KB
pacman.py	Python File	26 KB
pacmanAgents.py	Python File	3 KB
search.py	Python File	4 KB
searchAgents.py	Python File	21 KB
textDisplay.py	Python File	3 KB
util.py	Python File	26 KB

Följande filer ingår i projektet:

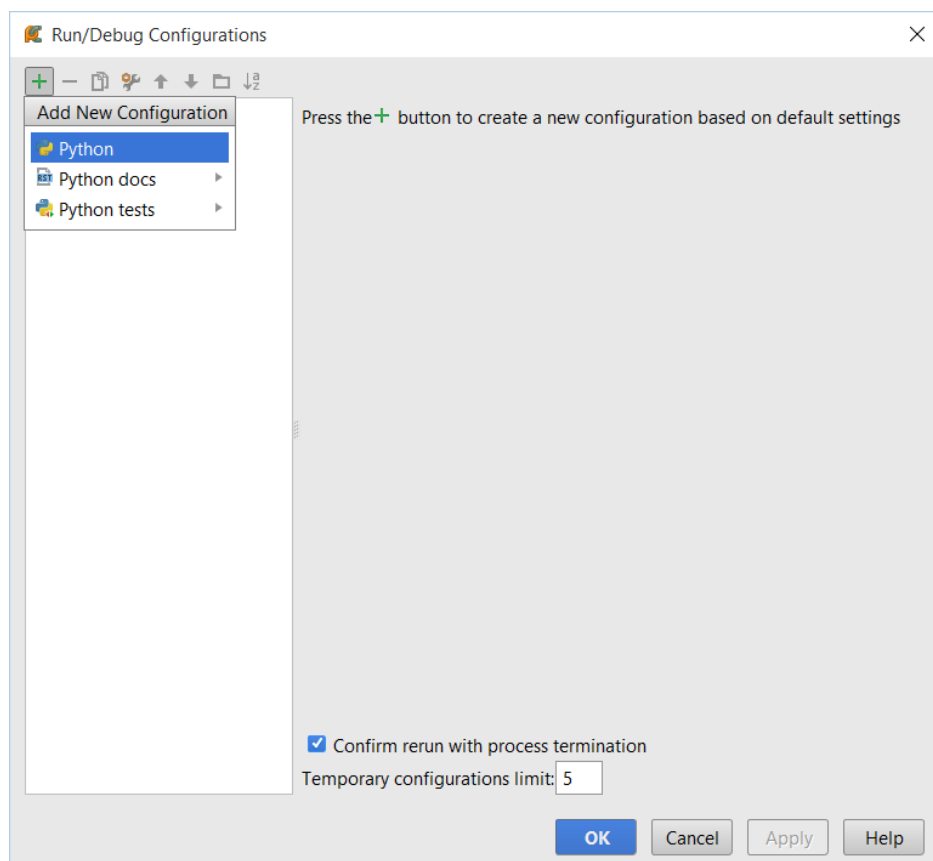
- `<layouts>` - folder som innehåller ett antal `.lay` filer (kartor, "levels" för Pacman)
- `eightpuzzle.py` - innehåller kod för "åtta"-spelet
- `game.py` - innehåller logiken för Pacman-spelet (innehåller bland annat klasserna `AgentState`, `Agent`, `Direction` och `Grid`)
- `ghostAgents.py` - Agenter som kontrollerar spöken
- `graphicsDisplay.py` - grafik för Pacman
- `graphicsUtils.py` - support för Pacman grafik
- `keyboardAgents.py` - Tangentbordsinterface för att kontrollera Pacman
- `layout.py` - kod för att läsa layout filer och för att lagra deras innehåll
- **`multiAgents.py`** - innehåller alla multi-agentbaserade sökagenter
- `pacman.py` - Huvudfilen som kör Pacman spel (innehåller `GameState` klassen som används i denna laborationen)
- `pacmanAgents.py` - innehåller några enkla exempelagenter för Pacman
- **`search.py`** - innehåller alla sökalgoritmer
- **`searchAgents.py`** - innehåller alla sökbaserade agenter
- `textDisplay.py` - ASCII grafik för Pacman
- `util.py` - användbara datastrukturer vid implementering av sökalgoritmer

De enda filerna som skall modifieras och lämnas in är filerna **`multiAgents.py`**, **`search.py`** och **`searchAgents.py`** (markerade med **fet stil**). Filerna `game.py`, `pacman.py` och `util.py` (markerade med *kursiv stil*) kan vara bra att läsa igenom. Övriga filer behöver ni inte bry er om.

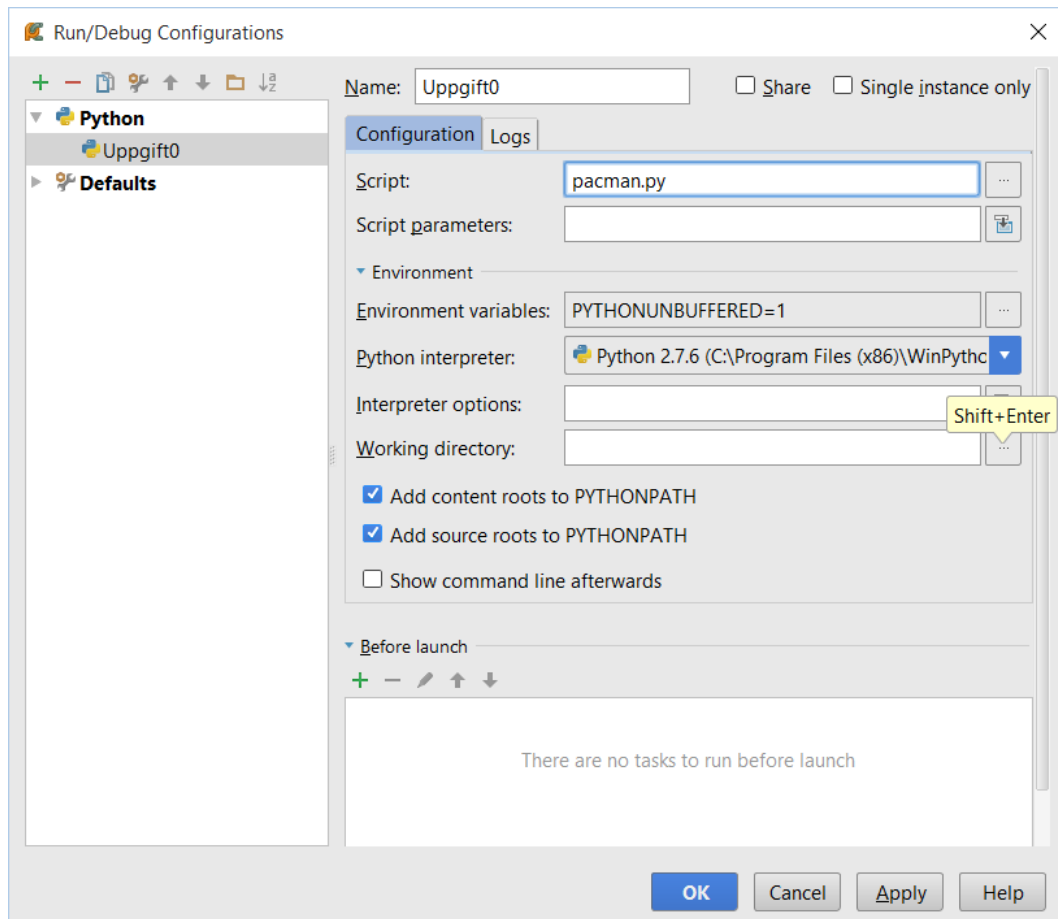
Klicka på pilen längst upp till höger i PyCharms GUI och välj **Edit Configurations**.



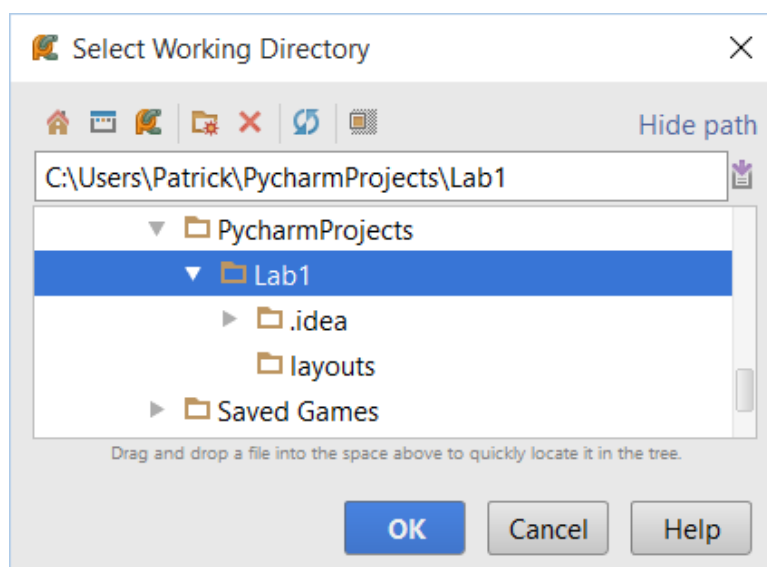
Klicka på "+"-knappen längst upp till vänster och välj **Python** från kontextmenyn.



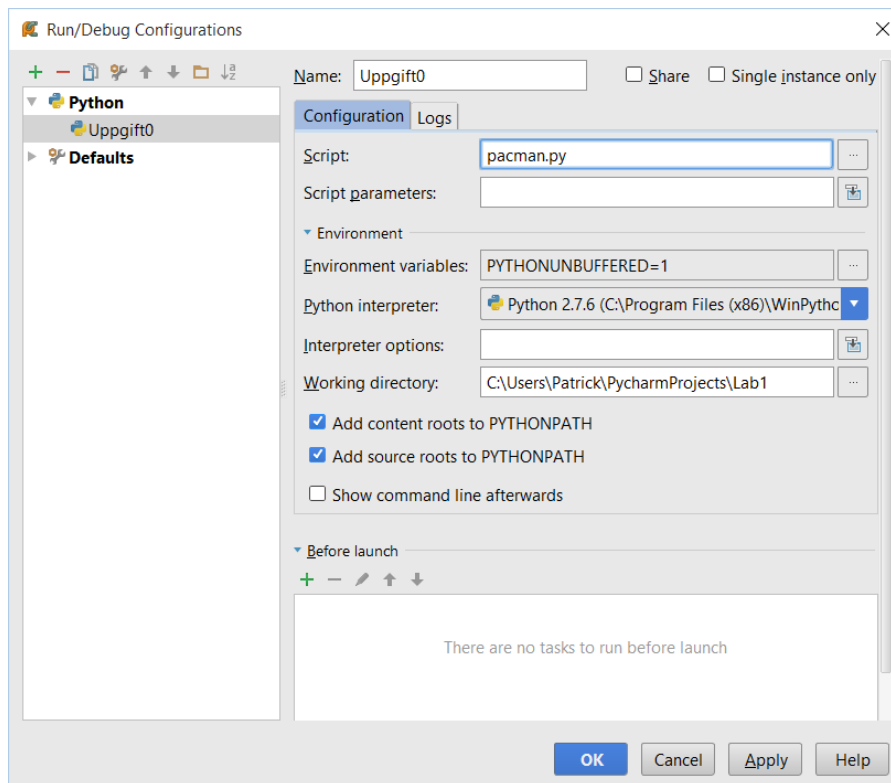
Fyll i "Uppgift0" i **Name** fältet, "pacman.py" i **Script** fältet och klicka på "..."-knappen jämte **Working directory** fältet (se nedanstående bild).



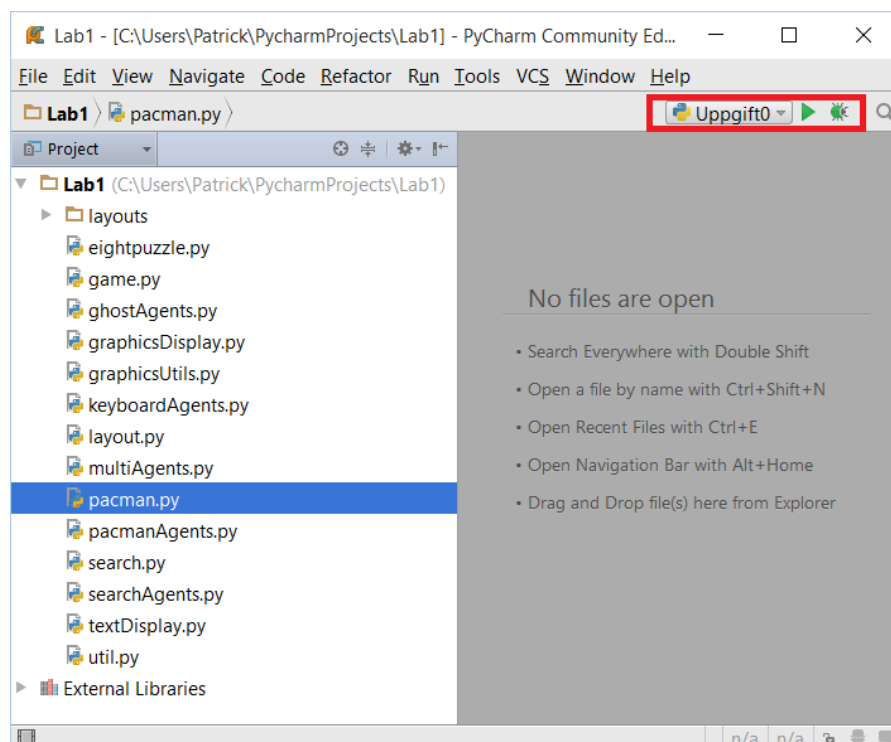
Se till så att din projektfolder är vald som din *Working Directory* och klicka på "OK"-knappen.



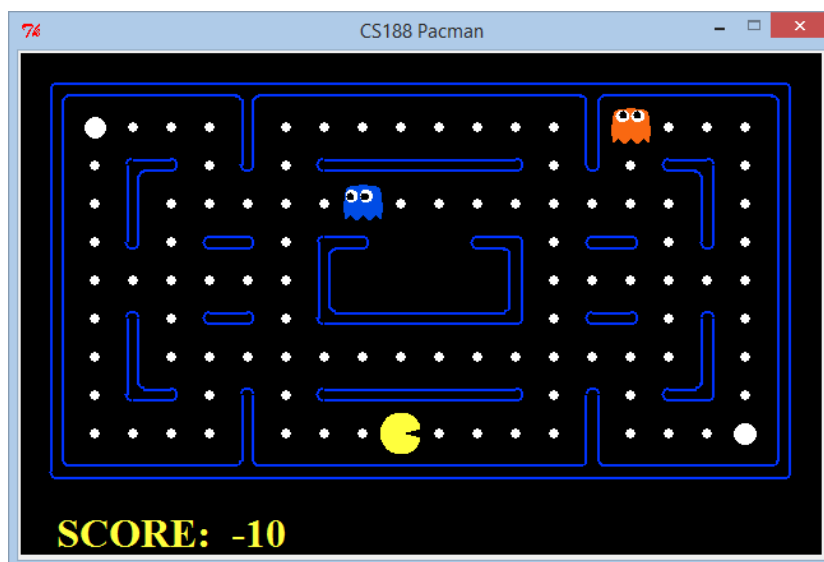
Klicka slutligen på "OK"-knappen i *Run/Debug Configurations* fönstret.



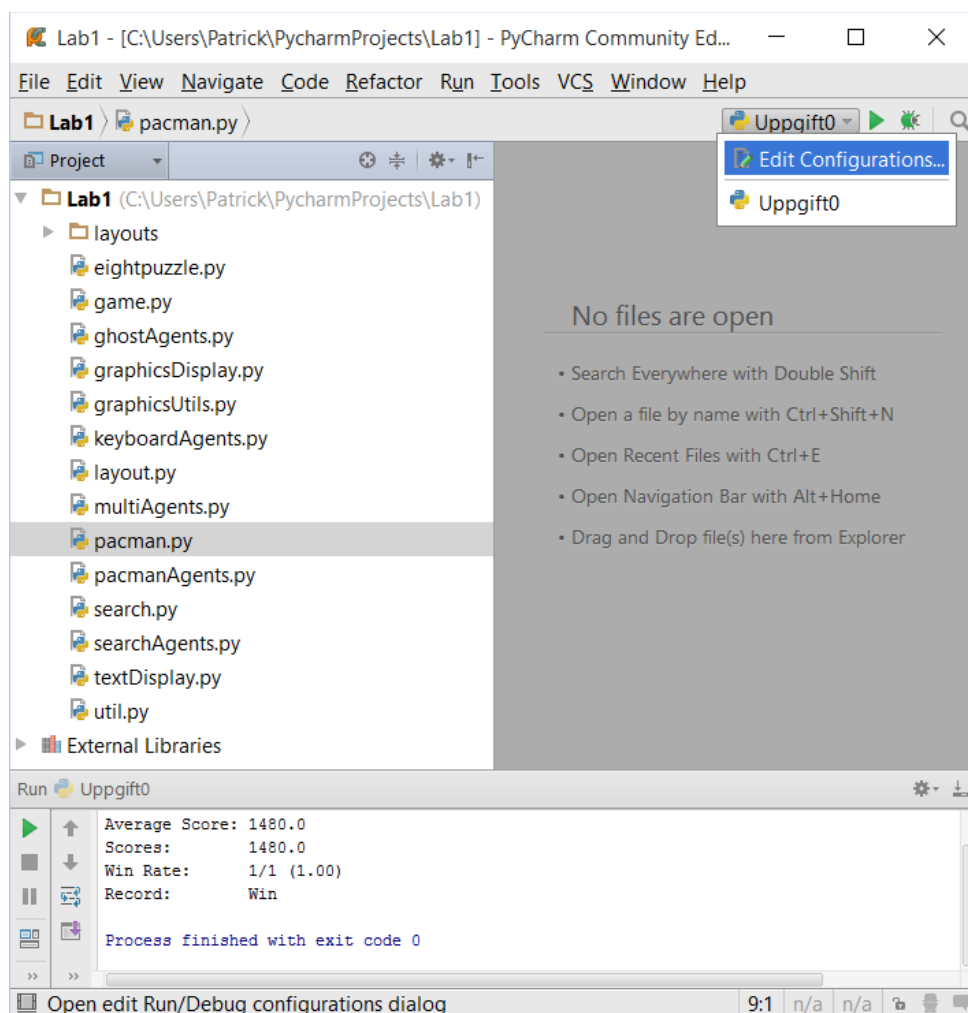
Du har nu skapat en *konfiguration* som kör/debuggar Pacman-spelet i normalt läge. Namnet på vald konfiguration ("Uppgift0") visas längst upp till höger i PyCharms GUI. För att testköra Pacman, klicka på den gröna högerpilen jämte namnet på vald konfiguration. För att debugga Pacman, klicka på den gröna "insekten" jämte den gröna högerpilen.



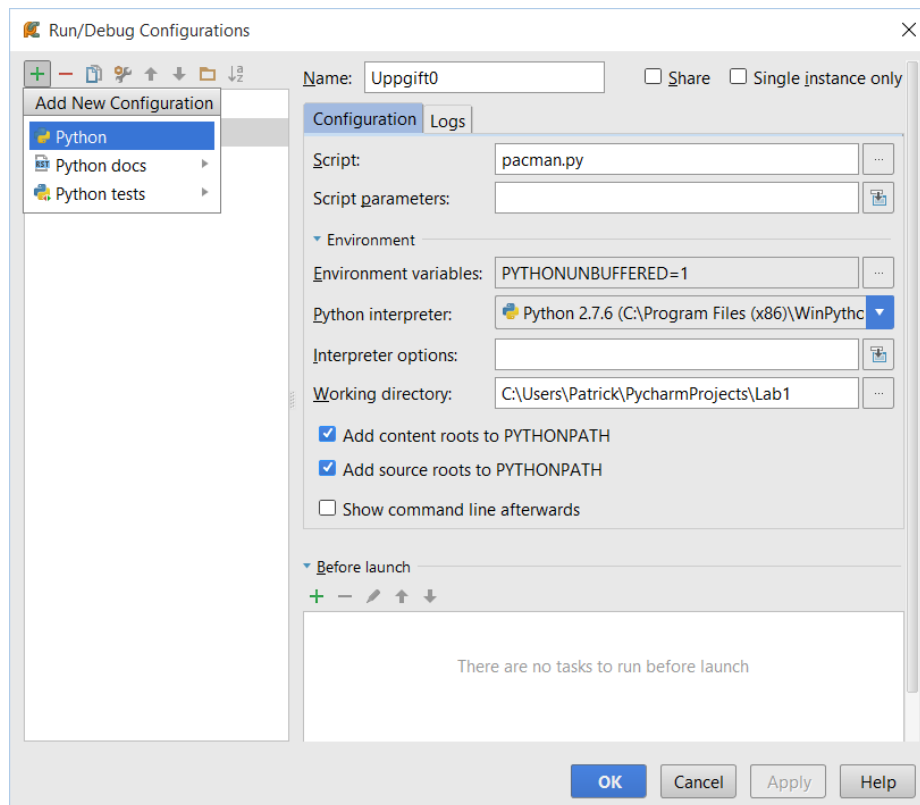
Pacman körs nu som ett vanligt spel, där du kan använda piltangenterna för att styra Pacman. Stäng fönstret för att avsluta Pacman-spelet.



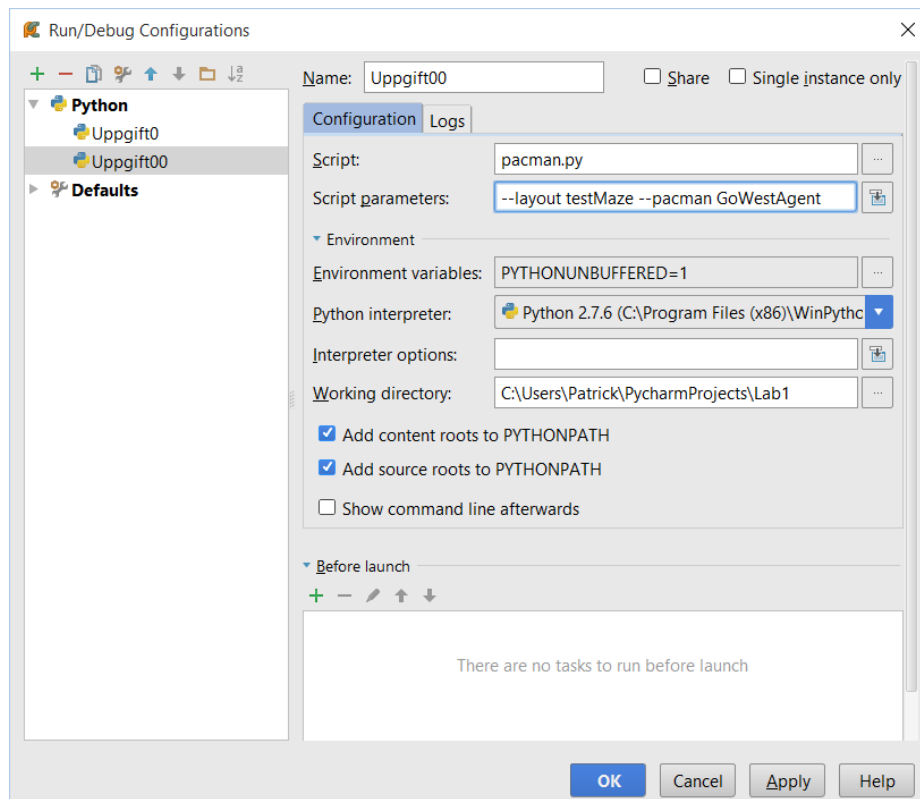
Klicka återigen på konfigurations-pilen längst upp till höger i PyCharms GUI och välj **Edit Configurations**.



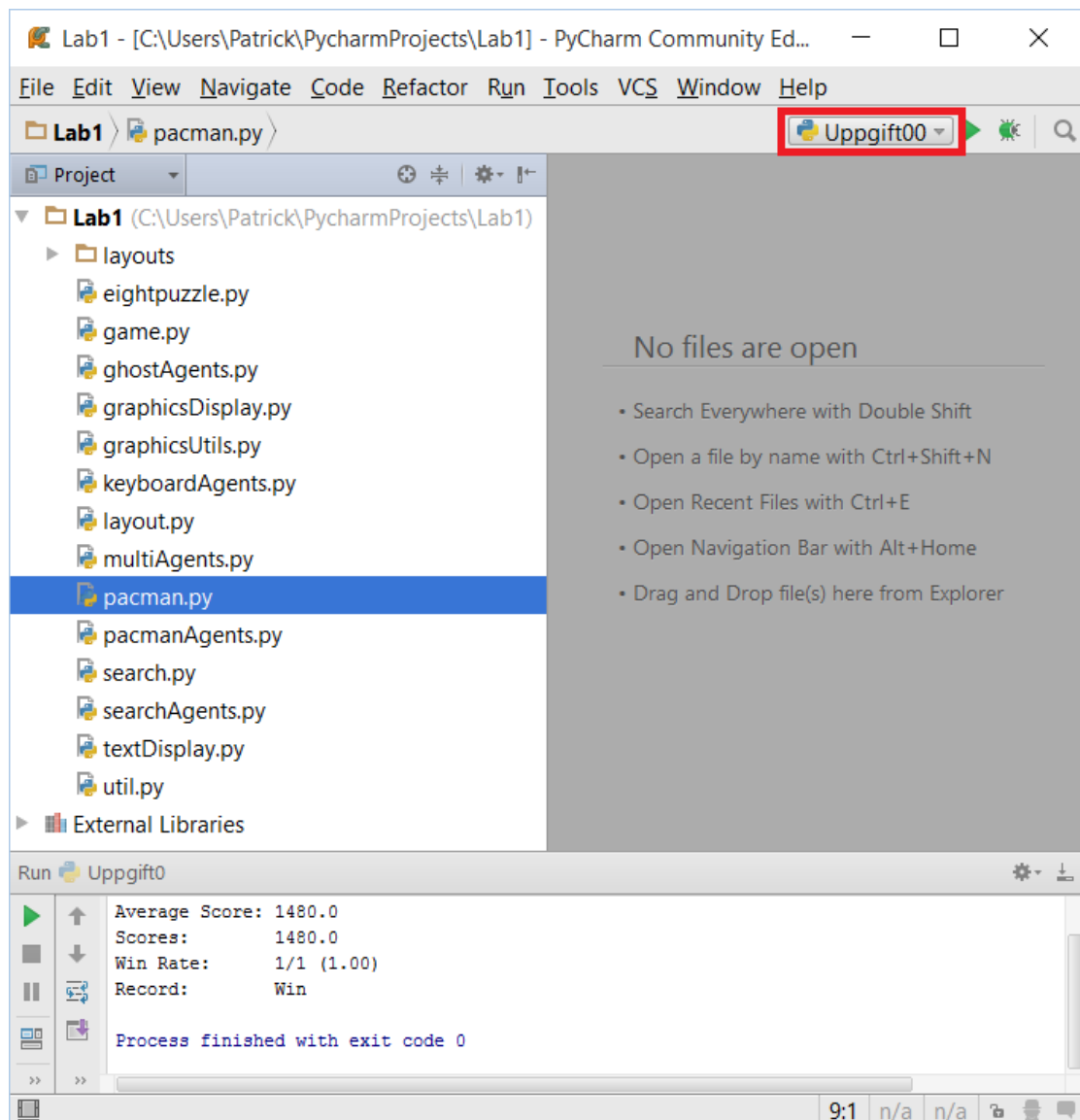
Klicka på "+"-knappen längst upp till höger och välj **Python**.



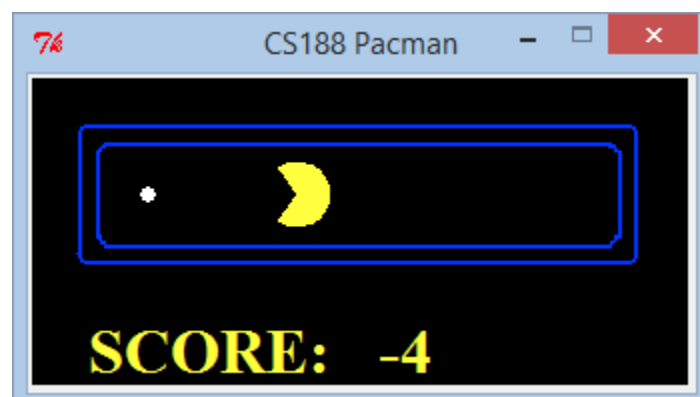
Fyll i "Uppgift00" i **Name** fältet, "pacman.py" i **Script** fältet och "--layout testMaze --pacman GoWestAgent" i **Script parameters** fältet. Se också till så att projektfoldern är vald i **Working directory** fältet. Klicka sedan på "OK"-knappen.



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



Pacman styrs nu av en Pacman-agent. Stäng fönstret för att avsluta Pacman-spelet.



Den nya konfigurationen körde pacman.py med följande växlar:

```
--layout testMaze --pacman GoWestAgent
```

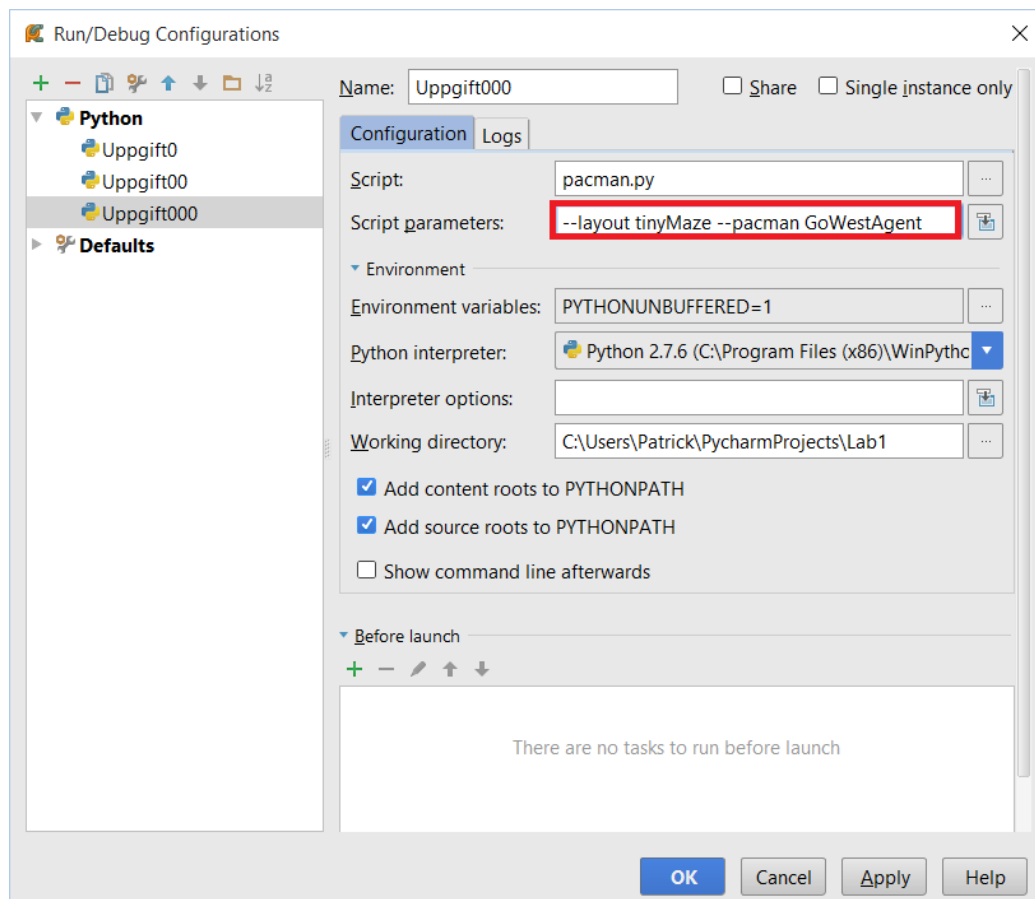
Dessa växlar ser till så att layouten med namn testMaze (filen testMaze.lay i foldern layouts) laddas och att Pacman-agenten med namn GoWestAgent (klassen GoWestAgent i filen searchAgents.py) används. I metoden GoWestAgent::getAction() ser man att denna agenten alltid kommer att välja handlingen "gå västerut" tills agenten springer in i en vägg.

```
class GoWestAgent(Agent):  
    "An agent that goes West until it can't."  
  
    def getAction(self, state):  
        "The agent receives a GameState (defined in pacman.py)."  
        if Directions.WEST in state.getLegalPacmanActions():  
            return Directions.WEST  
        else:  
            return Directions.STOP
```

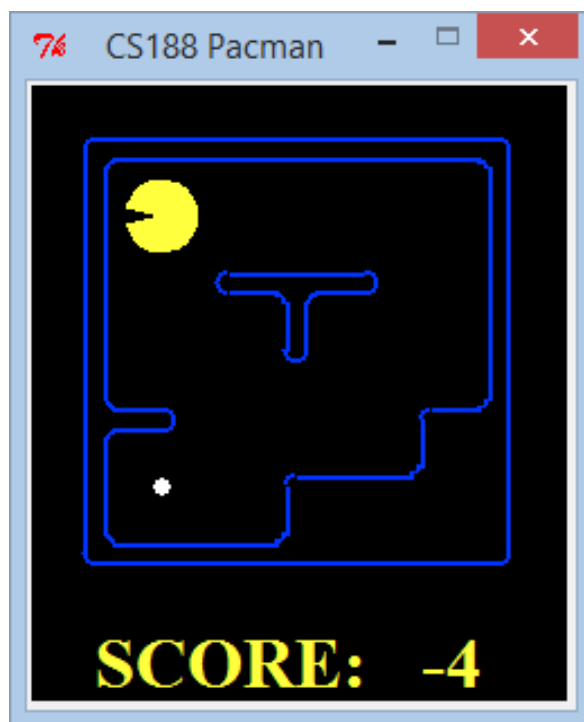
Skapa (på samma sätt som ovan) en ny konfiguration med namn "Uppgift000", men använd istället följande växlar till pacman.py:

```
--layout tinyMaze --pacman GoWestAgent
```

Den nya konfigurationen väljer endast en annan layout för Pacman.



Se till så att den nya konfigurationen är förvald och kör sedan Pacman igen i kör-läge eller debug-läge.



Denna gången fastar Pacman-agenten "GoWestAgent" i det övre vänstra hörnet, och eftersom poängen räknar ner med -1 för varje sekund så kommer denna "ointelligenta" agenten att få värsta möjliga poäng på just denna layouten. Stäng fönstret för att avsluta Pacman-spelet.

I nedanstående uppgifter skall problemlösande Pacman-agenter skapas som kan hantera en del layouts i foldern "layouts".

För varje deluppgift kommer en ny konfiguration att skapas med diverse växlar. Filen `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. T.ex. så är följande två rader ekvivalenta:

```
--layout tinyMaze --pacman GoWestAgent  
-l tinyMaze -p GoWestAgent
```

Nedan följer en beskrivning av samtliga växlar:

- `-n GAMES, --numGames=GAMES`
Antalet spel (GAMES) som skall köras (default=1). Denna växeln kommer främst att användas i laboration 2 då vi implementerar reinforcement learning.
- `-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 slumpalsgeneratorn (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 Uppgift 1 - Sökalgoritmer (singelagentmiljöer)

I filen **searchAgents.py** hittar ni klassen **SearchAgent** som ärver från klassen **Agent** (i filen **game.py**). **SearchAgent** implementerar en generisk problemlösande agent som först söker efter en plan (en sekvens av handlingar) i Pacmans värld från ett starttillstånd till ett måltillstånd och sedan utför handlingarna i planen steg för steg. Klassen innehåller en konstruktor **__init__()** och de två metoderna **registerInitialState()** samt **getAction()**.

Konstruktern accepterar tre inparametrar; en sökalgoritm, ett sökproblem samt en heuristisk funktion. Defaultparametrarna för dessa tre inparametrarna är "*depthFirstSearch*", "*PositionSearchProblem*" samt "*nullHeuristic*", dvs om dessa tre inparametrar inte specificeras explicit då konstruktern anropas, kommer en problemlösande agent att skapas, som utför en *djupet-först sökning* på problemet "*PositionSearchProblem*" utan någon *heuristik*. De två defaultparametrarna *depthFirstSearch* och *nullHeuristic* refererar till motsvarande två metoder i filen **search.py** medans defaultparametern *PositionSearchProblem* refererar till motsvarande klass i filen **searchAgents.py**.

Metoden **registerInitialState()** accepterar ett **GameState** objekt (definierad i **pacman.py**) som inparameter, skapar en instans av det valda problemet, använder den valda sökalgoritmen för att söka efter en plan (en sekvens av handlingar) och sparar undan planen i klassvariabeln **actions**. Med andra ord, så utförs all problemlösning då denna metoden anropas.

Metoden **getAction()** accepterar också ett **GameState** objekt som inparameter och returnerar nästa handling i planen (som finns lagrad i klassvariabeln **actions**).

I filen **search.py** finns metoderna **depthFirstSearch()** och **nullHeuristic()** som motsvarar två av konstrukterns tre defaultparametrar. Metoden **nullHeuristic()** accepterar en instans av ett tillstånd samt en probleminstans som inparametrar och returnerar endast värdet 0 (dvs ingen heuristik används). Metoden **depthFirstSearch()** accepterar en probleminstans som inparameter och är för närvarande inte implementerad. Ovanför metoden **depthFirstSearch()** finns dock ett exempel på en implementerad sökalgoritm i metoden **tinyMazeSearch()**. Denna "sökalgoritmen" kommer alltid att returnera handlingssekvensen [s, s, w, s, w, w, s, w], där "s" står för gå söderut och "w" står för gå västerut. Metoden ger ett exempel på hur samtliga sökalgoritmer arbetar, dvs de accepterar en probleminstans och returnerar en plan (sekvens av handlingar). Under metoden **depthFirstSearch()** finns också de oimplementerade metoderna **breadthFirstSearch()**, **uniformCostSearch()** och **aStarSearch()**.

Längst upp i filen **search.py** finns den abstrakta klassen **SearchProblem** som definierar den grundläggande strukturen för samtliga sökproblem. Klassen innehåller de fyra abstrakta metoderna **getStartState()**, **isGoalState()**, **getSuccessors()** och **getCostOfActions()**. För att definiera ett specifikt problem, ärver man från denna klassen och implementerar de fyra metoderna.

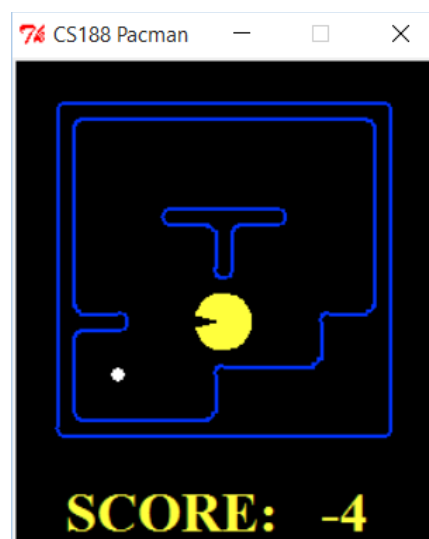
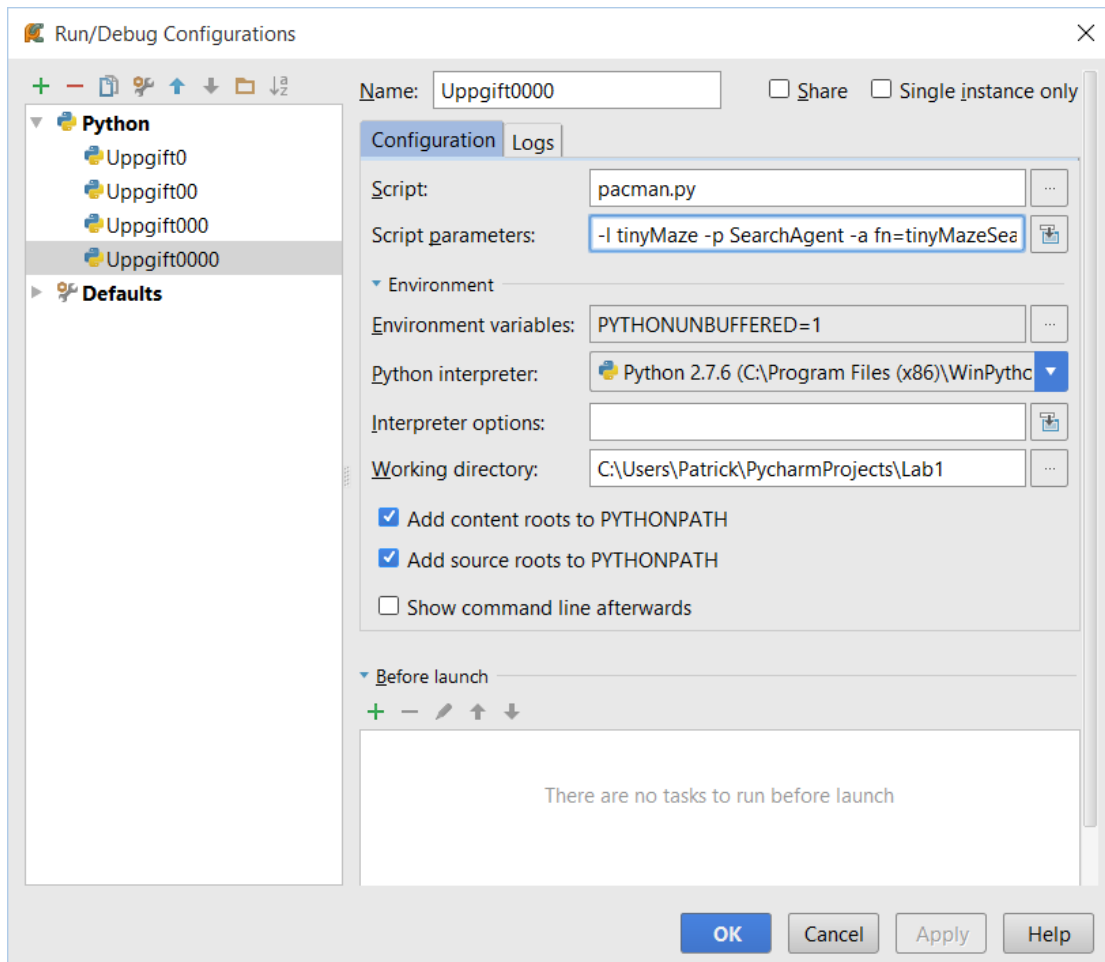
Metoden **getStartState()** skall returnera sökproblemets starttillstånd. Metoden **isGoalState()** accepterar ett söktillstånd som inparameter och returnerar ett booleskt värde som indikerar om söktillståndet utgör ett måltillstånd eller inte. Metoden **getSuccessors()** accepterar också ett söktillstånd som inparameter och returnerar en lista med samtliga direkt efterföljande söktillstånd som kan nås från nuvarande söktillstånd. Varje söktillstånd i listan består av en 3-tupel: (*successor*, *action*, *stepCost*). Elementet *successor* representerar ett efterföljande tillstånd som kan nås från nuvarande tillstånd, elementet *action* innehåller handlingen som behöver utföras för att komma till det efterföljande tillståndet och elementet *stepCost* utgör kostnaden för att gå till det efterföljande tillståndet. Metoden **getCostOfActions()** accepterar en lista med handlingar (en plan) som inparameter och returnerar den totala kostnaden för att utföra hela handlingssekvensen från starttillståndet till sluttillståndet.

I filen **searchAgents.py** finns defaultsökproblemet för den generiska problemlösande agenten definierad i klassen **PositionSearchProblem**. Denna klassen ärver från klassen **SearchProblem** (i filen **search.py**) och implementerar de fyra abstrakta metoderna samt en konstruktor. Klassen implementerar sökproblemet av att hitta en sökväg från nuvarande ruta till en specifik målruta i Pacman spelet, dvs tillståndsrymden består av (x,y) positioner (2-tupler).

Konstruktorn accepterar ett **GameState** objekt, en kostnadsfunktion, ett måltillstånd ett starttillstånd och två parametrar som styr varningsmeddelanden samt visualisering (dessa två sista parametrar behöver ni inte bry er om). Samtliga parametrar, utom den första, har defaultvärden (den konstanta kostnaden 1, måltillståndet (1,1), nuvarande tillstånd som starttillstånd, samt med varningsmeddelanden och visualisering påslagna). De fyra metoderna är triviala implementeringar av basklassens abstrakta metoder.

Skapa och kör nedanstående Python konfiguration (som **Uppgift0000**) för att testa den generiska problemlösande agenten **SearchAgent** med den triviala sökalgoritmen **tinyMazeSearch** på sökproblemet **PositionSearchProblem** med Pacman layouten **tinyMaze.lay**:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```



Sökalgoritmen som har implementerats i metoden **tinyMazeSearch()** fungerar endast för Pacman layouten **tinyMaze.lay**. Dock kan vi lösa många olika sökproblem med de generiska sökalgoritmerna djupet-först, bredden-först, uniform kostnad och A*. Speciellt, kan vi undvika problemet med återupprepade tillstånd i sökträdet genom att implementera grafsökningsvarianten ("fringe" + "closed" mängd) av de ovanstående fyra sökalgoritmerna. Slutligen vet vi att det som skiljer de fyra sökalgoritmerna åt är hur de hanterar söknoderna på "fringen", dvs datastrukturen som används för fringen.

3.1 Grafsökning (GraphSearch)

Skapa en metod **graphSearch()** i **search.py** som implementerar den generiska grafsökningsalgoritmen. Pseudokoden för grafsökningsalgoritmen kan hittas i föreläsningsslidesen (föreläsning F04), alternativt ges en variant på pseudokoden nedan.

```

function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)

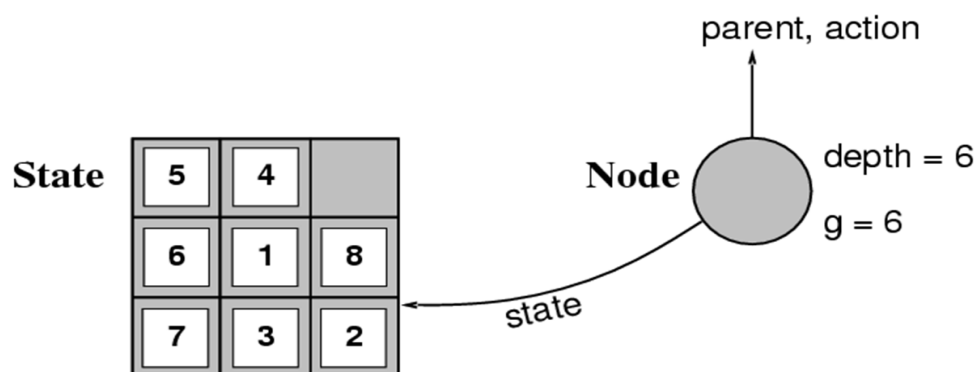
```

```

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors

```

Observera i pseudokoden för *EXPAND* att en söknod behöver innehålla flera attribut (se slidesen från föreläsning F03), vilket återges i bilden nedan. Skapa därför klassen **Node** i **search.py** som kan lagra dessa attributen.



Testa så att **graphSearch()** fungerar med djupet-först sökalgoritmen i uppgift 3.2.

3.2 Djupet-Först Sökning (Depth-First Search)

Implementera metoden **depthFirstSearch()** i filen **search.py**. OBS! Metoden skall utföra en djupet-först sökning genom att anropa **graphSearch()** med lämplig datastruktur (använd en av datastrukturerna längst upp i filen **util.py**). Ni behöver endast infoga kod där ni hittar texten `*** YOUR CODE HERE ***` (med eventuella *import* satser i början av filen **search.py**).

Testa så att du hittar en lösning till problemen med följande Python konfigurationer:

- `pacman.py -l tinyMaze -p SearchAgent -a fn=dfs`
- `pacman.py -l mediumMaze -p SearchAgent -a fn=dfs`
- `pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=dfs`

Om Pacman rör sig för långsamt, lägg till växeln `--frameTime 0` till konfigurationen.

Pacman-brädet visar vilka tillstånd som besökts genom att färga motsvarande rutor med en röd färg (ett tillstånd som besöks tidigare än ett annat har en ljusare färg). Kontrollera så att sökmönstret stämmer överens med din djupet-först sökning. Lägg också märke till att Pacman inte kommer att gå till varje söktillstånd då den valda handlingsplanen exekveras.

Med en korrekt implementerad **graphSearch()** metod och en korrekt vald datastruktur i **depthFirstSearch()** skall längden (kostnaden) för lösningen till *mediumMaze* vara 130 steg (med högst 146 expanderade noder och en "score" på 380).

3.3 Bredden-Först Sökning (Bredth-First Search)

Implementera metoden **breadthFirstSearch()** i filen **search.py**. OBS! Metoden skall utföra en bredden-först sökning genom att anropa **graphSearch()** med lämplig datastruktur (använd en av datastrukturerna längst upp i filen **util.py**). Ni behöver endast infoga kod där ni hittar texten `*** YOUR CODE HERE ***` (med eventuella *import* satser i början av filen **search.py**).

Testa så att du hittar en lösning till problemen med följande Python konfigurationer:

- `pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
- `pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5`

Återigen, om Pacman rör sig för långsamt, lägg till växeln `--frameTime 0`.

Med en korrekt implementerad **graphSearch()** metod och en korrekt vald datastruktur i **breadthFirstSearch()** skall längden (kostnaden) för lösningen till *mediumMaze* vara 68 steg (med högst 269 expanderade noder och en "score" på 442).

Dessutom, om du har skrivit generell sökkod skall du kunna återanvända samma sökkod till 8-pusslet. Om din sökkod är generell, skall du kunna köra nedanstående (växellösa) konfiguration för att lösa 8-pusslet (tryck <Enter> för att stega igenom lösningen):

`eightpuzzle.py`

3.4 Uniform Kostnad Sökning (Uniform Cost Search)

Implementera metoden **uniformCostSearch()** i filen **search.py**. OBS! Metoden skall utföra en uniform kostnadssökning genom att anropa **graphSearch()** med lämplig datastruktur (använd en av datastrukturerna längst upp i filen **util.py**). Infoga kod där ni hittar texten `*** YOUR CODE HERE ***` (med eventuella *import* satser i början av filen **search.py**). Observera att ni även måste definiera en kostnadsfunktion.

Testa så att du hittar en lösning till problemet med följande Python konfiguration:

- `pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`

Återigen, om Pacman rör sig för långsamt, lägg till växeln `--frameTime 0`.

Med en korrekt implementerad **graphSearch()** metod, en korrekt vald datastruktur i **uniformCostSearch()** och en korrekt kostnadsfunktion skall längden (kostnaden) för lösningen till *mediumMaze* vara 68 steg (med högst 269 expanderade noder och en "score" på 442), dvs samma som för bredden-först sökningen ovan.

3.5 A* Sökning (A* Search)

Implementera metoden **aStarSearch()** i filen **search.py**. OBS! Metoden skall utföra en A* sökning genom att anropa **graphSearch()** med lämplig datastruktur (använd en av datastrukturerna längst upp i filen **util.py**). Infoga kod där ni hittar texten `*** YOUR CODE HERE ***` (med eventuella *import* satser i början av filen **search.py**). Observera att ni även måste definiera en kostnadsfunktion.

Till skillnad från de övriga tre sökmetoderna accepterar **aStarSearch()** ytterligare en inparameter, *heuristic*. Defaultvärdet för denna parametern är *nullHeuristic*, dvs den heuristiska funktionen (metoden) **nullHeuristic()** i filen **search.py** kommer att användas som default om ingen annan heuristisk funktion specificeras explicit. Metoden **nullHeuristic()** visar också att en heuristisk funktion (i Pacman kodbasen) accepterar två inparametrar; *state* och *problem*. När du implementerar **aStarSearch()** metoden kan du dock använda metoden **manhattanHeuristic()** som redan finns implementerad i filen **searchAgents.py**.

Testa så att du hittar en lösning till problemet med följande Python konfiguration:

- `pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

Återigen, om Pacman rör sig för långsamt, lägg till växeln `--frameTime 0`.

Med en korrekt implementerad **graphSearch()** metod, en korrekt vald datastruktur i **aStarSearch()** och en korrekt kostnadsfunktion skall kostnaden för lösningen till *bigMaze* vara 210 (med högst 549 expanderade noder och en "score" på 300).


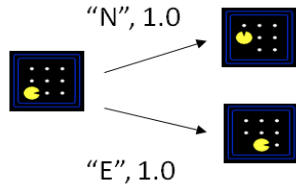


4 Uppgift 2 - Problemrepresentation

Sökproblemet i uppgift 1 bestod av att hitta en väg (handlingsplan) från Pacmans starttillstånd till måltillståndet som utgjordes av ruta (1,1). Detta sökproblem var redan definierat i klassen **PositionSearchProblem** i filen **searchAgents.py**.

Uppgift 2 består av att representera (definiera) problemet av att hitta en väg som går från pacmans starttillstånd och som besöker (går igenom) samtliga fyra hörn i Pacman brädet. Detta sökproblem skall definieras i klassen **CornersProblem** i filen **searchAgents.py**. Klassen ärver från klassen **SearchProblem** (i filen **search.py**) och behöver alltså implementera de fyra abstrakta basmetoderna **getStartState()**, **isGoalState()**, **getSuccessors()** och **getCostOfActions()** samt en konstruktor **__init__()** som accepterar ett **GameState** objekt (definierad i **pacman.py**).

Ni behöver endast infoga kod där ni hittar texten `*** YOUR CODE HERE ***` i **searchAgents.py**. Dessutom är metoden **getCostOfActions()** redan implementerad för er. I princip behöver ni initialisera problemet i **__init__()** (t.ex. hur starttillståndet ser ut för detta problemet), returnera starttillståndet i **getStartState()**, kolla ifall måltillståndet har nåtts i **isGoalState()** samt beräkna och returnera en lista med de efterföljande tillstånden som kan nås från nuvarande tillstånd i **getSuccessors()**.

Enligt slidesen från föreläsning F03, består ett sökproblem av:

- En tillståndsrymd 
- En övergångsfunktion med
 - Handlingar $(s_t, a_t) \rightarrow s_{t+1}$
 - Kostnader $c(s_t, a_t, s_{t+1})$
- Ett starttillstånd
 - $S_0 =$ 
- Ett måltest
 - Explicit: $S_t ==$  ?
 - Implicit: `SchackMatt(S_t)` ?

Metoderna **getStartState()** och **isGoalState()** motsvarar alltså de två sista punkterna ovan medans **getSuccessors()** och **getCostOfActions()** motsvarar den andra punkten (övergångsfunktionen), där **getSuccessors()** returnerar en lista med de omedelbart efterföljande tillstånden som kan nås via giltiga handling från ett specifikt tillstånd och **getCostOfActions()** beräknar den totala kostnaden för en hel sekvens av handlingar. Hela tillståndsrymden bildas då implicit via övergångsfunktionen **getSuccessors()**.

För att testa problemrepresentationen kommer bredden-först sökning att användas. Se därför till så att denna sökalgoritmen (från uppgift 1) fungerar problemfritt innan ni löser denna uppgiften.

4.1 CornersProblem

Komplettera klassen **CornersProblem** i filen **searchAgents.py** genom att definiera sökproblemet att ta sig från ett starttillstånd som går via de fyra hörnen på Pacman brädet (oavsett om det finns en "mat" prick där eller inte). Observera att de fyra hörnen för en specifik layout redan är definierade i konstruktorn **__init__()** med nedanstående tupel (som i sin tur innehåller fyra 2-tupler):

```
self.corners = ((1,1), (1,top), (right, 1), (right, top))
```

Tänk på att den valda tillståndsrepresentationen måste innehålla tillräckligt med information för att upptäcka ifall måltillståndet har nåtts. Dessutom behövs tillräckligt med information för att kunna skapa samtliga efterföljande tillstånd i metoden **getSuccessors()**. Dock skall inte irrelevant information tas med i ett söktillstånd, t.ex. skall definitivt inte ett **GameState** objekt användas som ett söktillstånd. Den enda informationen ni behöver om ett *speltillstånd* är i princip Pacmans startposition och de fyra hörnens positioner. Uppgiften blir inte godkänd om ni tar med irrelevant information för att representera ett *söktillstånd*.

Testa så att ni hittar en lösning till problemet med följande Python konfigurationer:

- `pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`
- `pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`

Om Pacman rör sig för långsamt, lägg till växeln `--frameTime 0` till konfigurationen.

Den optimala lösningen till *tinyCorners* layouten har en kostnad på 28 steg (med högst 252 expanderade noder för en bredden-först sökning och en "score" på 512).

Den optimala lösningen till *mediumCorners* layouten har en kostnad på 106 steg (med högst 1966 expanderade noder för en bredden-först sökning och en "score" på 434).

5 Uppgift 3 - Design av Heuristiska Funktioner

I uppgift 2 användes bredden-först sökning för att lösa **CornersProblem**. Vi kan dock använda A* sökalgoritmen med en lämplig heuristisk funktion för att snabba upp sökningen betydligt (dvs, minska ner på antalet expanderade noder). Eftersom A* sökalgoritmen kommer att användas för att lösa denna uppgiten, se först till så att A* sökalgoritmen har implementerats felfritt i Uppgift 1.

En *heuristik* är en funktion som tar ett *söktillstånd* som inparameter och returnerar ett tal som estimerar kostnaden till närmsta måltillstånd (i denna kodbasen accepterar dock varje heuristisk metod två inparametrar; söktillståndet *state* och probleminstansen *problem*). En effektivare heuristik returnerar ett värde (estimerad kostnad) som ligger närmre den verkliga kostnaden. För att en heuristik skall vara *admissibel* måste den returnera icke-negativa värden som inte överestimerar (dvs, måste vara mindre än eller lika med) det kortaste avståndet (lägsta kostnaden) till det närmsta måltillståndet. För att en heuristik skall vara *konsekvent* måste den vara admissibel och dessutom måste följande gälla: om den verkliga kostnaden är k för att utföra en handling i ett tillstånd A som resulterar i tillståndet B , så får minskningen i det heuristiska värdet för att gå från tillstånd A till tillstånd B inte vara större än värdet k (se föreläsning F04).

I den generiska *trädsökningsalgoritmen* är en admissibel heuristik ett tillräckligt villkor för att garantera en komplett sökalgoitm. Dock, i den generiska *grafsökningsalgoritmen* (vilken vi använder i denna laborationen) är inte en admissibel heuristik ett tillräckligt villkor för att garantera kompletthet. I detta fallet behöver vi det striktare kravet på en konsekvent heuristik. Oftast är en admissibel heuristik också konsekvent, speciellt om heuristiken fås genom att förenkla det ursprungliga problemet (detta är dock inte garanterat). Det lättaste sättet att ta fram en konsekvent heuristik är därför att först ta fram en väl fungerande admissibel heuristik genom att förenkla det ursprungliga problemet och sedan kontrollera att heuristiken också är konsekvent. Det enda sättet att garantera att en heuristik är konsekvent är med ett matematiskt bevis. Dock kan man lätt visa att en heuristik är inkonsekvent på empirisk väg genom att, för varje nod som expanderas, kontrollera så att samtliga efterföljande noders f -värden är minst lika stor (\geq) som den ursprungliga nodens f -värde. Kom ihåg att kostnaden för en nod n är $f(n) = g(n) + h(n)$ där $g(n)$ är kostnaden att gå från startnoden till noden n och $h(n)$ är det heuristiska värdet, dvs den estimerade kostnaden att gå från noden n till närmsta målnod. Dessutom, om *uniform kostnad sökning* och *A* sökning* (som använder samma $g(n)$ funktion) returnerar olika totala kostnader för ett och samma problem så är den heuristiska funktionen $h(n)$ inkonsekvent.

Det finns två triviala heuristiska funktioner; $h(n) = 0$ samt $h(n) = k^*$, dvs en heuristisk funktion som alltid returnerar värdet 0 (detta är samma sak som *uniform kostnad sökning*) samt en heuristisk funktion som returnerar den verkliga kostnaden k^* . Den heuristiska funktionen $h(n) = 0$ kommer inte innebära någon vinst i ett mindre antal expanderade noder medans den heuristiska funktionen $h(n) = k^*$ kommer att ta alldeles för lång tid att beräkna för icke-triviala problem. Den ultimata heuristiska funktionen, för ett specifikt problem, är en heuristisk funktion som minimerar söktiden (genom att skapa en perfekt balans mellan $h(n) = 0$ samt $h(n) = k^*$).

5.1 **cornersHeuristic**

Implementera en icke-trivial, konsekvent, heuristisk funktion i metoden **cornersHeuristic()** (i filen **searchAgents.py**) för sökproblemet **CornersProblem**.

Testa den heuristiska funktionen med följande Python konfiguration:

```
pacman.py
-l mediumCorners
-p SearchAgent
-a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Om Pacman rör sig för långsamt, lägg till växeln *--frameTime 0* till konfigurationen.

Den optimala lösningen till *mediumCorners* har en kostnad på 106 steg med en "score" på 434, precis som med bredden-först sökning i uppgift 4. Dock är vi mer intresserade av antalet expanderade noder i denna uppgiften. Följande tabell kan användas som referens på den heuristiska funktionens kvalitet:

Antal expanderade noder	Heuristikens kvalitet
> 2000	Dålig
1600 - 2000	Medelmåttig
1200 - 1600	Bra
< 1200	Mycket bra

OBS! För att få godkänt på denna uppgiften måste den heuristiska funktionen vara konsekvent!

6 Uppgift 4 - Sökalgoritmer (multiagentmiljöer)

De ovanstående uppgifterna behandlade *singelagentmiljöer*. De återstående uppgifterna behandlar *multiagentmiljöer*, dvs där andra agenter antingen försöker motverka eller samarbeta med eran agent. I Pacman-spelet försöker dock samtliga andra agenter (spöken) motverka eran agent (Pacman).

Ni kommer att implementera de *adversarial search* algoritmer som vi har gått igenom (föreläsning F05) för att hjälpa Pacman att lösa problemet med att äta alla "mat" prickar och samtidigt undvika samtliga spöken. Pacman har också tillgång till "Power Pellets", dvs de stora runda tabletterna som gör samtliga spöken sårbara under en viss tid.

Från föreläsningarna vet ni att **Minimax** algoritmen fungerar bra mot en motspelare som spelar *optimalt* i ett "*zero-sum game*", dvs mot en annan *rationell agent* som försöker maximera sin egna **nyttofunktion** (utility function), som är motstridig till eran agents nyttofunktion. I ett triviale problem, där tillståndsrymden inte är alltför stor, kan vi utforska samtliga vägar från ett visst tillstånd till varje **måltillstånd**, läsa av vårans agents **nyttovärde** (utility value) och "backa upp" nyttovärdet för det bästa måltillståndet till rotnoden. På så sätt blir det triviale att välja den bästa möjliga handlingen i varje tillstånd, dvs handlingen med högst nyttovärde för vårans agent (MAX noden i sökträdet).

De flesta verkliga problemen är dock icke-triviala, varför sökningen måste avbrytas vid ett visst **sökdjup** i sökträdet. Eftersom man inte har tillgång till ett exakt nyttovärde vid ett visst djup då sökningen avbryts innan ett måltillstånd har hittats, måste en *evalueringsfunktion* användas för att **estimera ett nyttovärde** för agenten. Denna funktionen brukar oftast utgöras av en **viktad linjär evalueringsfunktion**. En sådan funktion returnerar ett tal som bildas genom att summera ihop ett antal viktiga egenskaper för ett tillstånd, t.ex. *Avstånd_till_närmsta_spöke* - *Antal_mat_prickar_kvar*. Dessutom, för att få med hur viktig en viss egenskap är i relation till andra egenskaper, så multiplicerar man varje egenskap med en vikt, t.ex. $0.3 * \text{Avstånd_till_närmsta_spöke} - 0.5 * \text{Antal_mat_prickar_kvar}$. Ju högre värde som returneras av den viktade linjära evalueringsfunktionen desto bättre.

Vi kan också använda **alpha-beta pruning** för att minimera antalet expanderade noder i vårt sökträd. Alpha-värdet anger det bästa nyttovärdet längst en väg till rotnoden för vårans agent (**MAX spelaren**) medans beta-värdet anger det bästa nyttovärdet längst en väg till rotnoden för den andra agenten (**MIN spelaren**).

Från föreläsning F05 vet ni att Minimax algoritmen endast fungerar bra mot motspelare som spelar *optimalt*, men mindre bra mot andra typer av motspelare, t.ex. motspelare som använder någon slags slump i deras strategi. För att modellera en motspelare som utför helt slumpmässiga handlingar används istället **Expectimax** algoritmen. I den enklaste formen av Expectimax algoritmen utgörs eran agent som vanligt av en MAX nod i sökträdet medans motspelaren utgörs av en slumpnod (*chance node*), där ett viktat medelvärde av nyttovärden bildas från samtliga möjliga handlingar. Dessutom kan man använda flera MIN noder i Minimax sökträdet samt flera slumpnoder i Expectimax sökträdet om problemet innehåller fler än en motspelare.

Den abstrakta basklassen **Agent** för samtliga agenter i Pacman-spelet finns definierad i filen **game.py**. Basklassen innehåller en abstrakt metod **getAction()** som måste implementeras av samtliga ärvda agenterklasser. Metoden **getAction()** accepterar ett **GameState** objekt (definierad i filen **pacman.py**) och måste returnera en handling (**NORTH, SOUTH, EAST, WEST, STOP**) som finns definierad i klassen **Directions** (i **game.py**)

6.1 Reflexagent

Så att ni kan bekanta er med de attribut och metoder som finns i **GameState** objektet, börjar vi med att förbättra den redan implementerade enkla reflexagenten **ReflexAgent** som finns definierad längst upp i filen **multiAgents.py**. **ReflexAgent** ärver från **Agent** och implementerar basklassens abstrakta metod **getAction()**. Metoden hämtar först samtliga giltiga handlingar från **GameState** objektet och anropar sedan, för varje giltig handling, klassmetoden **evaluationFunction()** med **GameState** objektet och den giltiga handlingen som inparametrar. Metoden **evaluationFunction()** returnerar ett nyttovärde, som i detta fallet endast består av "score" för varje efterföljande tillstånd som Pacman-agenten hamnar i då respektive giltig handling har utförts, för varje handling. Därefter returneras handlingen med högst nyttovärde (om flera handlingar har samma bästa nyttovärde, slumpas ett av dessa handlingarna ut) från metoden **getAction()**. Studera dessa två metoderna och lägg märke till vilken information som finns tillgänglig i **GameState** objektet (kika också på **GameState** objektet i filen **pacman.py**).

Testa först reflexagenten (utan att modifiera koden) med nedanstående konfiguration:

- `pacman.py -p ReflexAgent -l testClassic`

Förbättra sedan reflexagenten genom att kombinera olika egenskaper som erhålls från **GameState** objektet i **evaluationFunction()** metoden. För att erhålla en bra reflexagent måste åtminstone både positionen av mat, **getFood()**, samt spöken, **getGhostStates()**, beaktas i evalueringsfunktionen. Den förbättrade reflexagenten skall kunna äta upp alla mat-prickar i ovanstående Python konfiguration (*testClassic*) utan några problem.

Testa också reflexagenten med nedanstående Python konfigurationer:

- `pacman.py --frameTime 0 -p ReflexAgent -l mediumClassic -k 1`
- `pacman.py --frameTime 0 -p ReflexAgent -l mediumClassic -k 2`

Reflexagenten kommer ofta att dö med två spöken (-k 2), om inte eran reflexagents evalueringsfunktion är riktigt bra. Om ni vill använda samma slumpvisa sekvens av handlingar för spöket (för att utvärdera olika inställningar för evalueringsfunktionen) så kan ni använda växeln -f. Ni kan också köra flera spel i sekvens genom att använda växeln -n <antal spel>. Dessutom kan ni stänga av grafiken med växeln -q om ni vill köra flera spel i snabb sekvens.

Jag kommer att testa eran reflexagent på nedanstående Python konfiguration (för att bli godkända på uppgiften måste eran reflexagent vinna 3 av 10 spel när jag testar):

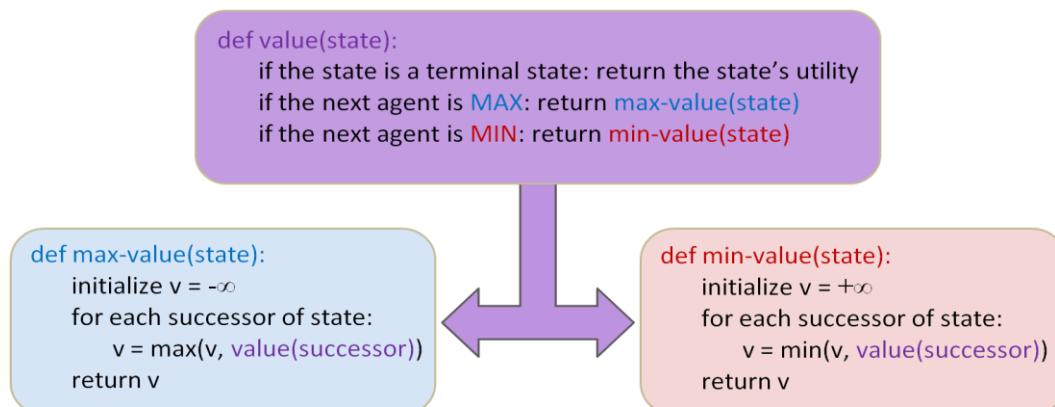
- `pacman.py --frameTime 0 -p ReflexAgent -l openClassic`

6.2 Minimax

Reflexagenten valde nästa handling genom att kika ett steg frammåt i tiden och beräkna ett estimerat nyttovärde för varje efterföljande tillstånd. I denna uppgiften skall ni implementera Minimax algoritmen i klassen **MinimaxAgent** som finns definierad i filen **multiAgents.py**. **MinimaxAgent** ärver ifrån klassen **MultiAgentSearchAgent** (i **multiAgents.py**) som i sin tur ärver från den abstrakta Agent klassen (i **game.py**).

Klassen **MultiAgentSearchAgent** är endast till för att definiera gemensamma attribut och metoder för samtliga multiagent klasser. Ni behöver inte lägga till något i denna klassen om ni inte vill men ta inte bort något från klassen. Tre viktiga attribut som sätts i denna klassens konstruktor **__init__()** är **index**, **evaluationFunction** och **depth**. Attributet **index** används för att slå upp en viss agent bland spelets alla agenter (Pacman + spöken). Pacman har alltid index 0 medans spöke 1-n har index 1-n. Attributet **evaluationFunction** anger vilken evalueringsfunktion som skall användas och har defaultvärdet *"scoreEvaluationFunction"*. Denna evalueringsfunktion, som finns definierad i metoden **scoreEvaluationFunction()** i **multiAgents.py**, accepterar ett **GameState** objekt och returnerar endast "score" för respektive **GameState**. Slutligen anger attributet **depth** det maximala sök djupet för sökalgoritmen. Varje djup i spelet består av att Pacman utför ett drag varpå varje spöke svarar med ett drag, dvs djupet 2 innebär att Pacman och varje spöke har utfört 2 drag var.

Klassen **MinimaxAgent** ärver från **MultiAgentSearchAgent** och har därför tillgång till de tre attributen **index**, **evaluationFunction** och **depth**. **MinimaxAgent** måste dock implementera basklassens (**Agent**) abstrakta metod **getAction()**. Förutom metoden **getAction()** skall ni implementera de tre metoderna **value()**, **maxValue()** och **minValue()** enligt pseudokoden för Minimax algoritmen enligt nedan.



Tänk på att om ett terminaltillstånd eller max sök djup uppnås skall evalueringsfunktionen returnera ett estimerat nyttovärde. Ni skall **inte** implementera en egen evalueringsfunktion för denna uppgiften, utan kommer att använda den redan implementerade metoden **scoreEvaluationFunction()**. Ni behöver alltså endast koncentrera er på att implementera Minimax algoritmen på ett korrekt sätt. Tänk speciellt på att ni behöver en MAX nod för Pacman efterföljt av en MIN nod för **VARJE** spöke (och det kan finnas flera spöken för vissa Pacman layouter). Dessutom skall eran kod kunna söka till ett arbiträrt sök djup (djupet som skickas in som växel till **pacman.py**).

Täk på att:

- en korrekt implementerad Minimax agent fortfarande kommer att förlora en del spelomgångar eftersom agenten endast söker till ett visst djup, där ett estimerat nyttovärde från evalueringsfunktionen returneras
- Pacman alltid har **index** 0 och samtliga spöken har ett **index** från 1 till n
- alla tillstånd i Minimax skall vara **GameState** objekt (som inparameter till **getAction()** eller som skapas via **GameState.generateSuccessor()**)
- i större layouter såsom *openClassic* och *mediumClassic* kommer Pacman att irra omkring en hel del utan att göra några framsteg (t.ex. så kan pacman gå fram och tillbaka mellan två tillstånd jämte en mat-prick utan att äta den) men detta gör inget eftersom detta kommer att fixas till i uppgift 6.5.
- med Minimax algoritmen kommer Pacman att försöka avsluta spelet så fort som möjligt om han anser att han kommer dö eftersom "score" dekrementeras med 1 för varje diskret tidssteg

Testa Minimax-agenten med nedanstående Python konfigurationer:

- `pacman.py -p MinimaxAgent -l minimaxClassic -a depth=1`
- `pacman.py -p MinimaxAgent -l minimaxClassic -a depth=2`
- `pacman.py -p MinimaxAgent -l minimaxClassic -a depth=3`
- `pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4`

Minimax värdet för initialtillståndet i ovanstående konfigurationer skall vara 9, 8, 7 och -492 för sök djup 1, 2, 3 respektive 4.

6.3 AlphaBetaAgent

Minimax agenten fungerar bra om tillståndsrymden inte är alltför stor. Dock kan vi både snabba upp algoritmen och hantera större tillståndsrymder om vi implementerar alpha-beta pruning.

Implementera Minimax algoritmen med alpha-beta pruning i klassen **AlphaBetaAgent** (i **multiAgents.py**). Tänk på att alpha-beta pruning algoritmen måste fungera för flera agenter (spöken). Använd nedanstående pseudokod för MAX och MIN när ni implementerar alpha-beta pruning.

α : MAX's best option on path to root
 β : MIN's best option on path to root

```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize v =  $-\infty$   
    for each successor of state:  
        v = max(v, value(successor,  $\alpha$ ,  $\beta$ ))  
        if v >  $\beta$  return v  
         $\alpha$  = max( $\alpha$ , v)  
    return v
```

```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize v =  $+\infty$   
    for each successor of state:  
        v = min(v, value(successor,  $\alpha$ ,  $\beta$ ))  
        if v <  $\alpha$  return v  
         $\beta$  = min( $\beta$ , v)  
    return v
```

Testa agenten med nedanstående Python konfigurationer:

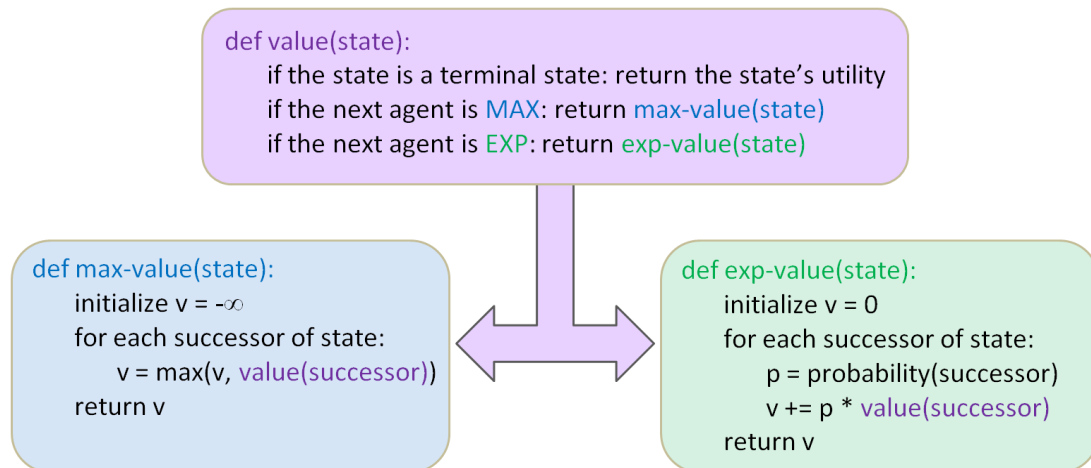
- pacman.py -p AlphaBetaAgent -l minimaxClassic -a depth=1
- pacman.py -p AlphaBetaAgent -l minimaxClassic -a depth=2
- pacman.py -p AlphaBetaAgent -l minimaxClassic -a depth=3
- pacman.py -p AlphaBetaAgent -l minimaxClassic -a depth=4

Minimax värdet för initialtillståndet i ovanstående konfigurationer skall vara 9, 8, 7 och -492 för sökdjup 1, 2, 3 respektive 4. Dvs, ni skall få precis samma värden som för Minimax agenten, dock skall sökningen gå mycket snabbare (ett mindre antal noder skall expanderas).

6.4 ExpectimaxAgent

Minimax algoritmen med alpha-beta pruning fungerar alldeles utmärkt om motspelaren spelar *optimalt*. Dock, i alla andra fall, är det bättre att modellera motspelaren med stokastiska handlingar. Vi kan använda **Expectimax** algoritmen för att åstadkomma detta.

Implementera Expectimax algoritmen i klassen **ExpectimaxAgent** (i filen **multiAgents.py**) med nedanstående pseudokod som utgångspunkt (se också föreläsningsslidesen från föreläsning F05).



Ni kan anta att motspelaren väljer bland sina giltiga drag med likformig sannolikhet, dvs skapa helt enkelt medelvärdet av motspelarens giltiga handlingars nyttovärden. **OBS!** Tänk på att ni använder rella tal när ni beräknar era medelvärden! Heltalsdivision i Python kommer att trunkera era medelvärden, dvs $1/2 = 0$, medans samma beräkning med rella tal blir korrekt, dvs $1.0/2.0 = 0.5$.

Testa agenten med nedanstående Python konfigurationer:

- `pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=1`
- `pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=2`
- `pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3`
- `pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=4`

Jämför Expectimaxagenten med AlphaBeta agenten genom att köra nedanstående Python konfigurationer (10 spel utan grafik med sökdeep 3):

- `pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10`
- `pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10`

Om Expectimax algoritmen är korrekt implementerad skall ExpectimaxAgent vinna cirka hälften av sina spel (medans AlphaBetaAgent skall förlora samtliga sina spel).

6.5 **betterEvaluationFunction**

Slutligen skall ni implementera en bättre evalueringsfunktion än **scoreEvaluationFunction()** som hittills har använts i era multiagentsökalgoritmer.

Implementera en bättre evalueringsfunktion i metoden **betterEvaluationFunction()** längst ner i filen **multiAgents.py**. Precis som i reflexagentens evalueringsfunktion kan en bättre evalueringsfunktion erhållas genom att kombinera olika egenskaper som kan fås från **GameState** objektet. Om ni dessutom viktar de olika egenskaperna och summera ihop dem erhåller ni en viktad linjär evalueringsfunktion.

Jag kommer att testa eran evalueringsfunktion med nedanstående Python konfiguration (för att bli godkända måste eran agent vinna 3 av 10 spel när jag testar):

- `pacman.py -p ExpectimaxAgent -l smallClassic -a depth=2,evalFn=better -q -n 10`

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 1. 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 18 november i rum L433.
- Handledning 2 sker måndagen den 23 november i rum L433.
- Handledning 3 sker torsdagen den 26 november i rum L433.

7.2 Inlämning

Laborationen lämnas in via PingPong, under Aktivitet/Inlämning Lab 1, där **endast** filerna **search.py**, **searchAgents.py** samt **multiAgents.py**, lämnas in som **en** arkivfil (7z, rar, tar eller zip).

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 29 november 23:59.
- Deadline för inlämningstillfälle 2 är onsdagen den 9 december 23:59.
- Deadline för inlämningstillfälle 3 är söndagen den 28 augusti 23:59.

7.3 Betyg

Endast betygen Underkänd eller Godkänd förekommer på laborationen.