COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2019

[Assignment 2]

Solving Pac-Man:
online Graph Search

Handed out: Thursday, 10 of October
Due: 00:00, Thursday, 24 of October

Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with dynamic memory allocation and your understanding of data structures, through programming a search algorithm over Graphs.
- Gain experience with applications of graphs and graph algorithms to solving games, one form of artificial intelligence.

Assignment description

In this programming assignment you'll be expected to build an *AI algorithm* to solve Pac-Man. The game invented in 1980 is one of the classics among arcade games. You can play the game compiling the code given to you using the keyboard, or using this web implementation.

The code in this assignment was adapted from the open-source terminal version made available by Mike Billars[1] and the original version can be installed as a standard package in Ubuntu[2].

The Pac-Man game

As explained in the wikipedia entry, The player navigates Pac-Man through a maze with no dead ends. The maze is filled with Pac-Dots, and includes four roving multi-colored ghosts: Blinky, Pinky, Inky, and Clyde.

The objective of the game is to accumulate as many points as possible by eating dots, fruits, and ghosts. When all of the dots in a stage are eaten, that stage is completed, and the player will advance to the next. The four ghosts roam the maze and chase Pac-Man. If any of the ghosts touches Pac-Man, a life is lost. When all lives have been lost, the game is over.

Pac-Man can eat a fruit first and then eat the ghosts for a fixed period of time to earn bonus points. The enemies turn deep blue, reverse direction and move away from Pac-Man, and usually move more slowly. When an enemy is eaten, its eyes return to the center ghost box where the ghost is regenerated in its normal color. The bonus score earned for eating a blue ghost increases exponentially for each consecutive ghost eaten while a single energizer is active: a score of 200 points is scored for eating one ghost, 400 for eating a second ghost, 800 for a third, and 1600 for the fourth.

The level id and a scoreboard can be found on the lower part. The information in the last three lines of the screen reveals information about the algorithm execution.

The game is won when all dots have been eaten. An AI agent or human player can change the direction of Pac-Man movements.

[1] https://sites.google.com/site/doctormike/pacman.html
[2] https://packages.ubuntu.com/xenial/games/pacman4console

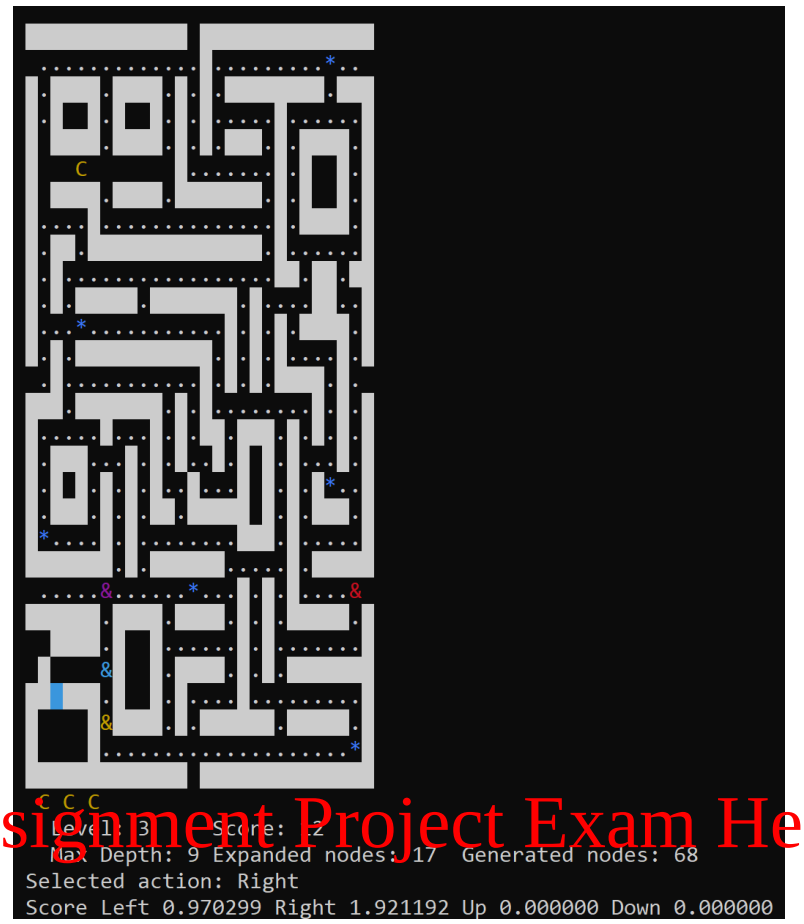

Figure 1: The UI of the Terminal version. c is pacman, & are ghosts, * are fruits, and . is a regular food.

The Algorithm

Each possible configuration of the Pac-Man game 29x28 grid and other relevant information such as the direction of pacman movements, number of lives left, etc. is called a *state*. The Pac-Man Graph $G = \langle V, E \rangle$ is implicitly defined. The vertex set V is defined as all the possible configurations (states), and the edges E connecting two vertexes are defined by the legal movements (right, left, up, down).

Your task is to find the path leading to the highest score, i.e. leading to the most rewarding vertex (state). A path is a sequence of movements. You are going to use a variant of Dijkstra to explore the most rewarding path first, up to a **maximum budget** B of expanded/explored nodes (nodes for which you've already generated its children).

Every time the game asks you for a movement (action), you should explore all possible paths until consuming the budget B if possible. Once you finished generating all the paths, you should return the **first action only** of the path leading to the highest score vertex. This action will then be executed by the game engine.

You might have multiple paths with the same maximum score. If more than one action (left,right,up or down) begins paths with the same maximum score, you'll have to break ties randomly.

Make sure you manage the memory well. Everytime you finish running the algorithm, you have to free all the nodes from the memory, otherwise you are going to run out of memory fairly fast or cause memory leaks.

```
GraphSearch(Graph, start, budget)
 1  node ← start
 2  explored ← empty Array
 3  frontier ← priority Queue Containing node Only
 4  while frontier ≠ empty
 5  do
 6      node ← frontier.pop()
 7      explored.add(node)
 8      if size(explored) < budget
 9        then
10              for each APPLICABLE action a ∈ {Left, Right, Up, Down}
11              do
12                  newNode ← applyAction(node)
13                  propagateBackScoreToFirstAction(newNode)
14                  if lostLife(newNode)
15                    then
16                          delete newNode
17                    else
18                          frontier.add(newNode)
19
20  freeMemory(explored)
21  bestAction ← select best action breaking ties randomly
22  return bestAction
```


Figure 2: Online Graph algorithm variant of Dijkstra

Every time that you consider all the actions that can be applied for a given node, only use the ones that will face Pac-Man towards a free tile. For example, in Figure 1 you should only consider the actions Left, and Right.

When you *applyAction* you have to create a new node, that

1. points to the parent,
2. updates the state with the action chosen,
3. updates the priority (used by the priority queue) of the node to be the negative node's depth d (if the node is the dth step of the path, then its priority is -d). This ensures the expansion of the shortest paths first, as the priority queue provided is a max heap;
4. updates the reward to be $r(n) = (h(n) + score(n) - score(nParent)) \times \gamma^d$
   (a) the heuristic value $h(n)$ that biases the reward to account for losing lives and eating fruits, plus
   (b) the change in score from the current node and the parent node
   (c) times a discount factor of $\gamma = 0.99^d$, where d is the depth of the node,
5. updates the accumulated reward from the initial node up to the current node, and
6. updates any other auxiliary data in the node.

The heuristic function is $h(n) = i - l - g$, where $i = 10$ if Pac-Man has eaten a fruit and becomes invincible in that state; $l = 10$ if a life has been lost in that state; and $g = 100$ if the game is over. Otherwise $i = l = g = 0$.

You are going to need some auxiliary data structures to update the scores of the first 4 applicable actions. The function *propagateBackScoreToFirstAction* takes the score of the newly generated node, and propagates back the score to the first action of the path.

This propagation can be either *Maximize* or *Average* :

- If you *Maximize*, you have to make sure that the first action is updated to reflect the maximum score of any of its children.
- If you *Average*, you have to make sure that the first action is updated to reflect the average score taking into account all its children.

Deliverables, evaluation and delivery rules

Deliverable 1 – *Solver* source code

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command: `make` generating an executable called `pacman`. Remember to compile using the optimization flag `gcc -O3` for doing your experiments, it will run twice faster than compiling with the debugging flag `gcc -g`. For the submission, please **submit your makefile with** `gcc -g` **option**, as our scripts need this flag for testing. Your program must not be compiled under any flags that prevents it from working under gdb or valgrind

Your implementation should work well over the standard levels, but might fail to perform well in some of our provided t_*.dat levels. The t_* levels are specifically designed to test challenging situations. Try different budgets and explain why your agent works well (or doesn't) in each of the test cases.

Base Code

You are given a base code. You can compile the code and play with the keyboard. The default solver chooses an action randomly. You are going to have to program your solver in the file ai.c. Look at the file pacman.c (MainLoop function) to know which function is called to select an action to execute.

You are given the structure of a node, the state, and also a priority queue implementation. Look into the utils.* files to know about the functions you can call to apply an action to update a game state.

You are free to change any file.

Input

You can play the game with the keyboard by executing

```
./pacman <level>
```

where level $\in \{0, \ldots, 9\}$ for standard levels, or the path to a `file.dat` level

In order to execute your solver use the following command:

```
./pacman <level> <ai/ai_pause> <max/avg> <budget>
```

Where `ai_pause` calls your algorithm and pauses the game to allow playing one step at a time. <max/avg> is either max or avg, to select the 2 options for propagating scores, and <budget> is an integer number indicating the budget of your search.

for example:

```
./pacman 3 ai max 1000
```

Will run max updates after expanding a 1000 nodes on `Levels/level03.dat` file.

```
./pacman LevelsTest/t_buridans_ass.dat ai_pause avg 1000
```


Will run average updates after expanding a 1000 nodes on `LevelsTest/t_buridans_ass.dat` file, and pause.

Output

Your solver will print into an `output.txt file` the following information:

1. Max Depth
2. Number of generated nodes.
3. Number of expanded nodes.
4. Number of expanded nodes per second.
5. Total Search Time, in seconds.
6. Maximum value in the board.
7. Score

For example, the output of your solver `./pacman ai max 1000` could be:

```
Propagation=Max
Budget=1000
MaxDepth = 8
TotalGenerated = 499,911
TotalExpanded = 253,079
Time = 7.05 seconds
Expanded/Second = 35,897
Score=543
```

These numbers are made up. MaxDepth stands for the node with maximum depth generated by your algorithm across the game. Expanded/Second can be computed by dividing the total number of expanded nodes by the time it took to play the game. A node is expanded if it was popped out from the priority queue, and a node is generated if it was created using the applyAction function. You can print all this information in the function `ExitProgram` in `pacman.c`

Deliverable 2 – Experimentation

Besides handing in the solver source code, you're required to provide a table with the mean score and deviation (if any), mean expanded/second and deviation, and total execution time for each type of propagation (max/avg) you implement and each max budget of 10,100,1000,2000.

In order to test your solver, you have to average over multiple runs because pacman has a random component: the movement of the ghosts. A sample of 3 runs is enough for the purpose of this assignment. Test at least 3 different Levels.

For each propagation type, plot a figure where the x axis is the budget, and y is the mean score.

Explain your results using your figures and tables. Which max/avg budget works best? Is it better to propagate max or avg?

If you do any of the optimizations for the extra mark, please report and discuss it in your experimentation.

Answer concisely. **Please include your Username, Student ID and Full Name** in your Document.

Evaluation

Assignment marks will be divided into three different components.

1. Solver (11)
2. Code Style (1)
3. Experimentation (3)

Please note that you should be ready to answer any question we might have on the details of your assignment solution by e–mail, or even attending a brief interview with me, in order to clarify any doubts we might have.

Code Style

You can improve the base code according to the guidelines given in the first assignments. Feel free to add comments wherever you find convenient. From your comments it should be very clear exactly which lines implement each line of the pseudocode. The base code was minimally modified to allow easy deployment of your AI algorithm and the coding style belongs to the original author.

Delivery rules

You will need to make *two* submissions for this assignment:

- Your C code files (including your `Makefile`) will be submitted through the LMS page for this subject: *Assignments* → *Assignment 2* → *Assignment 2: Code*.

- Your experiments report file will be submitted through the LMS page for this subject: *Assignments → Assignment 2 → Assignment 2: Experimentation*. This file can be of any format, e.g. .pdf, text or other.

Program files submitted (Code)

Submit the program files for your assignment and your `Makefile`.

Your programs *must* compile and run correctly on the JupyterHub server. You may have developed your program in another environment, but it still *must* run on the JupyterHub server at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on the JupyterHub server at reasonably frequent intervals.

`valgrind` will report a memory leak as **still reachable** memory. This is a known issue with `ncurses` and won't count as a leak. Make sure you maintain the other sections under **Leak summary** down to 0.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

To compile in your local machine you need to install the library `ncurses`. Ncurses is needed for the terminal UI. If you are using WSL or Ubuntu, type the following command:


```
sudo apt-get install libncurses5-dev libncursesw5-dev
```


Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

"Borrowing" of someone else's code without acknowledgement is plagiarism, e.x. taking code from a book without acknowledgement. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic honesty and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) on the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Administrative issues

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you decide to make a late submission, you should send an email directly to the lecturer as soon as possible and he will provide instructions for making a late submission.

What are the marks and the marking criteria Recall that this project is worth 15% of your final score. There is also a hurdle requirement: you must earn at least 15 marks out of a subtotal of 30 for the projects to pass this subject.

Finally Despite all these stern words, **we are here to help!** There is information about getting help in this subject on the LMS pages. Frequently asked questions about the project will be answered in the LMS discussion group.

Have Fun!

COMP20003 team.