

# Artificial Intelligence Coursework (COMS30084)

[Boiler-plate](#) • [Coursework Brief](#) • [GridWorld Library](#) • [Part 1](#) • [Part 2](#) • [Part 3](#) • [Part 4](#) • [Marking Scheme](#)

This document provides an overview of and guidelines for the **coursework assessment (COMS30084)** on the 3rd year **Artificial Intelligence (AI)** unit in the **2024-5** academic year. After outlining the standard boiler-plate text (Section 1) this document provides a general coursework brief (Section 2), followed by an introduction of the relevant features of the GridWorld library (Section 3), a breakdown of the four individual parts that you will need to submit (Section 4 - Section 7), and a summary of the marking scheme (Section 8).

## 1 Boiler-plate

---

**Value:** This coursework is for unit COMS30084 AI and is worth **70% of 20 (=14) credit points**.

**Mode:** This coursework is **individual work** and you must not discuss this task with anyone else (including your current classmates, former students, third parties, discussion forums, etc). Any collaboration will be considered cheating and will be treated as such, if detected.

**Timing:** The coursework will be released on Friday 8th November (Week 8) at 1pm and must be submitted by **Thursday 29th November** (Week 11) at 1pm. The intention is that you submit by **12pm** and keep the last hour as an emergency reserve for e.g. technical problems. In case of problems with your submission, e-mail [coms-student-enquiries@bristol.ac.uk](mailto:coms-student-enquiries@bristol.ac.uk) **before** the 1pm final deadline to avoid your work being counted as late.

**Submission:** You must submit the coursework on the **Blackboard** page for the assessment unit COMS30084 (not the teaching unit COMS30014). On the assessment unit, go to the menu item "Assessment, Submission and Feedback" and follow the instructions there.

**Commitment:** You are expected to work on both your courseworks in the 3-week period from Week 9 to Week 11 as if it were a working week in a regular job, that is 5 days a week for no more than 8 hours a day. It is up to you how you distribute your time and workload between the two units within those constraints. You are strongly advised not to try and work excessive hours during the coursework period: this is more likely to make your health worse than to make your marks better. If you need further pastoral/mental health support, please talk to your personal tutor, a senior tutor, or the university wellbeing service.

**Support:** For this unit COMS30084, the following support is available during the coursework period: we will continue running the lab as an optional **drop-in clinic** on Fridays at 11am-1pm. The lab will not be to help you complete the coursework, but to provide an opportunity to clarify what is expected of you in the coursework and also provide an opportunity to ask general questions about Prolog or any other part the unit (whether you are doing the major or minor); and you may also continue posting questions to the **Discussion Forum** on the Teaching Unit (COMS30014).

**Marking:** A summary of the marking scheme for this coursework is included in Section 8 of this document.

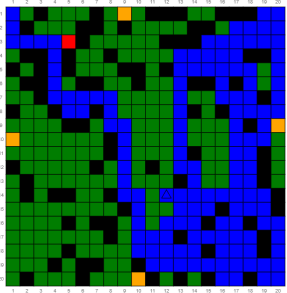
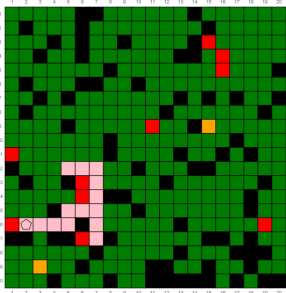
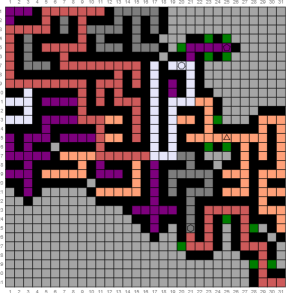
## 2 Coursework Brief

---

This coursework comprises **4 equally weighted parts** that are briefly outlined below (and explained in more detail in subsequent sections of this document):

- **Part 1 (25%)** involves you extending *Lab Search* with an A\* algorithm in order to more efficiently find **shortest paths** from an agent's position to specified cells or objects without running out of energy in grids containing walls, oracles and charging stations;
- **Part 2 (25%)** involves you extending Part 1 with the ability to plan more **complex routes** that enable your agent to visit as many oracles as possible, refuelling as necessary along the way, in order to find the identity of a secret film actor by asking each oracle to provide you with a single link from that actor's Wikipedia page;
- **Part 3 (25%)** involves you coordinating the movement of **multiple agents** in order to explore and solve an initially hidden **maze** by guiding each agent from an "entrance" point near the top left to an "exit" point at the bottom right;
- **Part 4 (25%)** is a more open-ended exercise that involves writing a **short report** where you consider (at a conceptual level rather than an implementation level) potential extensions of and reflections upon your solutions to the above tasks.

The following table provides a brief overview of the three coding Parts 1-3 showing the submission predicates and files (where 12345 stands for your **student number** - which should actually be 7 digits long) along with the associated grid configurations, a quick-start guide to running the code, and the relevant marking criteria:

Overview	Part 1	Part 2	Part 3
Solution predicate	solve_task/2	find_identity/1	solve_maze/0
Submission filename	cw_12345.pl	cw_wp_12345.pl	cw_maze_12345.pl
Grid size: $n$	20	15-25 (odd or even)	11-31 (odd only)
Visibility	initially <b>visible</b>	initially <b>visible</b>	initially <b>hidden</b>
# Agents	1	1	1-10
# Oracles	1	1-10	0
# Chargers	4	2-4	0
# Obstacles	~80	~80-100	$\sim n^2/2$
Position	start in top left	start at random	start in top left exit at bottom right
Energy	max=100 initial=100% but is <b>reduced</b> by 1 unit per <b>move</b> and 10 units per <b>query</b>	max= $n^2/4$ initial=50-100% but is <b>reduced</b> by 1 unit per <b>move</b> and max/10 per <b>query</b>	not applicable!
How to run <sup>†</sup>	./ailp.pl cw part1 start. . shell. setup. demo.	./ailp.pl cw part2 start. . shell. setup. identity.	./ailp.pl cw part3 start. . join_game(As,4). reset_game. start_game. solve_maze.
Example			
Task	Use A* search to solve tasks of form: find(Obj) ; go(Pos)	visit and query oracles to find secret actor id	lead all agents out of the maze

Marking Criteria (5 marks each)	<ul style="list-style-type: none"> <li>* minimising moves</li> <li>* minimising time</li> <li>* indirect paths</li> <li>* edge cases</li> <li>* good coding</li> </ul>	<ul style="list-style-type: none"> <li>* maximise oracles</li> <li>* minimising moves</li> <li>* minimising time</li> <li>* edge cases</li> <li>* good coding</li> </ul>	<ul style="list-style-type: none"> <li>* one agent</li> <li>* three agents</li> <li>* multi agents</li> <li>* end game</li> <li>* time taken</li> </ul>
--	--	--	---

**Table 1:** *Quick-start Guide for Parts 1-3.*



† Please note that you **MUST** hit "run" in the GridWorld **browser window** that will open (or nothing will happen until you do)! See later in this document for alternative ways of running the code in **Linux** and **Windows**



Remember to **rename** the default skeleton **submission files** above by **replacing** the number **12345** with your actual **student number** (a 7-digit number that is **not** the same as your **candidate number** or your **user id**)! Note that failure to use your student number will disrupt the loader and lead to seemingly obscure errors.

You are advised to tackle the parts **in the order they are listed above** as the level of guidance deliberately decreases as you progress through Parts 1-4. Moreover, Parts 2 and 3 both build upon the insights and code you develop in Part 1; and Part 4 will benefit from your experiences all the other Parts 1-3.



As there is no dependency between **Parts 2&3**, you may start work on either or both of these after you've made some **progress** on (but not necessarily completed) **Part 1**. Similarly, you may begin working on **Part 4** after making some **progress** on (but not necessarily completing) **Parts 1-3**.

This coursework **assumes** you have a **running version** of [SWI Prolog](#) (VERSION 8) and a **working knowledge** of *logic programs* obtained by working through **Chapters 1-6 & 10-11** of the recommended [Learn Prolog Now!](#) tutorial. You should have studied the content of the **Prolog I-IV lectures** along with **SWI toolchain and debugging slides** and worked through all three **Prolog labs** (in weeks 1,2 and 3) including reflecting on the **example solutions**. You should be comfortable editing and running programs in SWIPL as well as using the [online manual](#) and the built-in [debugger](#).



On some computers, it has been noticed that the latest version of SWI **may experience slow performance** due to changes in the http libraries (resulting in delays when interacting with cells next to oracles). If your machine is affected, please downgrade to [SWI v.8.4.3-1](#) — which is the version running on the lab machines (in MVB 2.11 and QB 1.80) that you may also use.



Note that, in this documentation, predicates are often annotated with their respective **arities** (name/arity) and arguments are often annotated with their intended **modes** (+In, -Out or ?Any) These **arity and mode decorations** should *not* be typed in any actual code - there are included in the documentation to show how the predicates should be used. As explained in the [SWI manual](#):

- an **input argument** (+) must be instantiated to a correctly typed term when the predicate is called,
- an **output argument** (-) will become instantiated (if it wasn't already) when the predicate succeeds,
- and **partial arguments** (?) may be variable, ground or partially instantiated when the predicate is called and/or succeeds.

Please note that (SWI built-in and GridWorld library) predicates are only guaranteed to work properly (or even at all) when used in the correct way!

You should treat this coursework as a **individual take-home open-book time-limited extended-exam**. You should work **by yourself** and without assistance from each other or TAs. But the following support will be available: we will continue running the Drop-In clinic on Fridays at 11am-1pm; and you may also continue posting questions to the discussion forum on the Teaching Unit (COMS30014). Please note that staff will be able to help clarify expectations and discuss general issues but they will **not** be allowed to explicitly help you write coursework solutions.



Please note that staff will be able to help clarify expectations and discuss general issues but will **not** be able to explicitly help you write coursework solutions.

The **deadline** for this coursework is 13:00 on Thursday (the 7th of December) of Week 11. Answers should be submitted in **Blackboard** by uploading exactly **4 files** (corresponding to the four respective parts) named as follows: `cw_12345.pl`, `cw_wp_12345.pl`, `cw_maze_12345.pl` and `cw_report_12345.pdf` (where 12345 should be replaced by your **student number**). Note that you may overwrite your files until the deadline - at which point standard Faculty **late penalties** will apply (to the submission as a whole). The relevant submission point should be available at least a week before the deadline.

Academic **offences** (including submission of work that is not your own, falsification of data/evidence or the use of materials without appropriate referencing) are all taken very seriously by the University. Suspected offences will be dealt with in accordance with the University's policies and procedures. If an academic offence is suspected in your work, you will be asked to attend an interview with senior members of the school, where you will be given the opportunity to defend your work. The plagiarism panel are able to apply a range of

penalties, depending the severity of the offence. These include: requirement to resubmit work, capping of grades and the award of no mark for an element of assessment.

If the completion of your assignment is significantly disrupted by serious health conditions, personal problems, or other similar issues, you may be able to apply for consideration of **extenuating circumstances** (in accordance with the normal university policy and processes).

### 3 GridWorld Library

---

A **Zip file** containing the **GridWorld library files** and **skeleton submission files** should be downloaded from Blackboard and unzipped on your working machine. These files extend and **supersede** the library code from the earlier **Prolog labs** (which means that many of the exported predicates should already be familiar to you). The following diagram provides an overview of the **key dependencies** between **Parts 1-4** of the coursework, the **user predicates** they'll need to define, the **library predicates** they'll need to call, and the **files** where everything is or needs to be located. Note that:

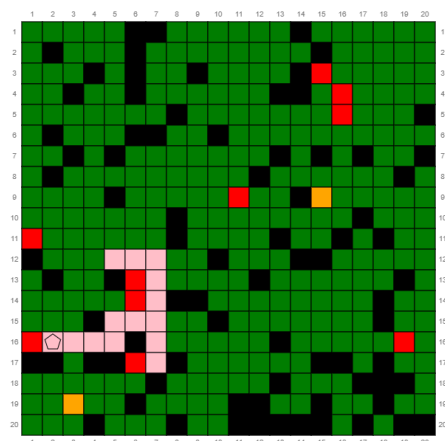
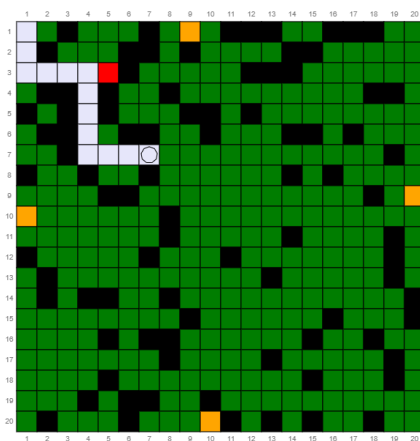
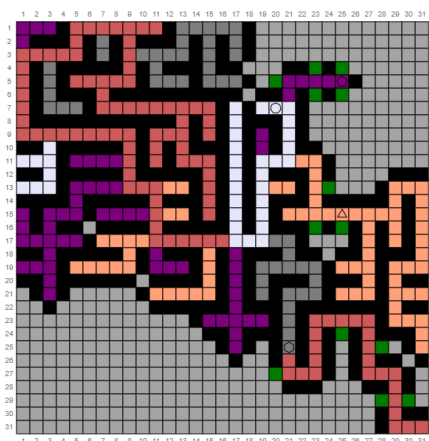
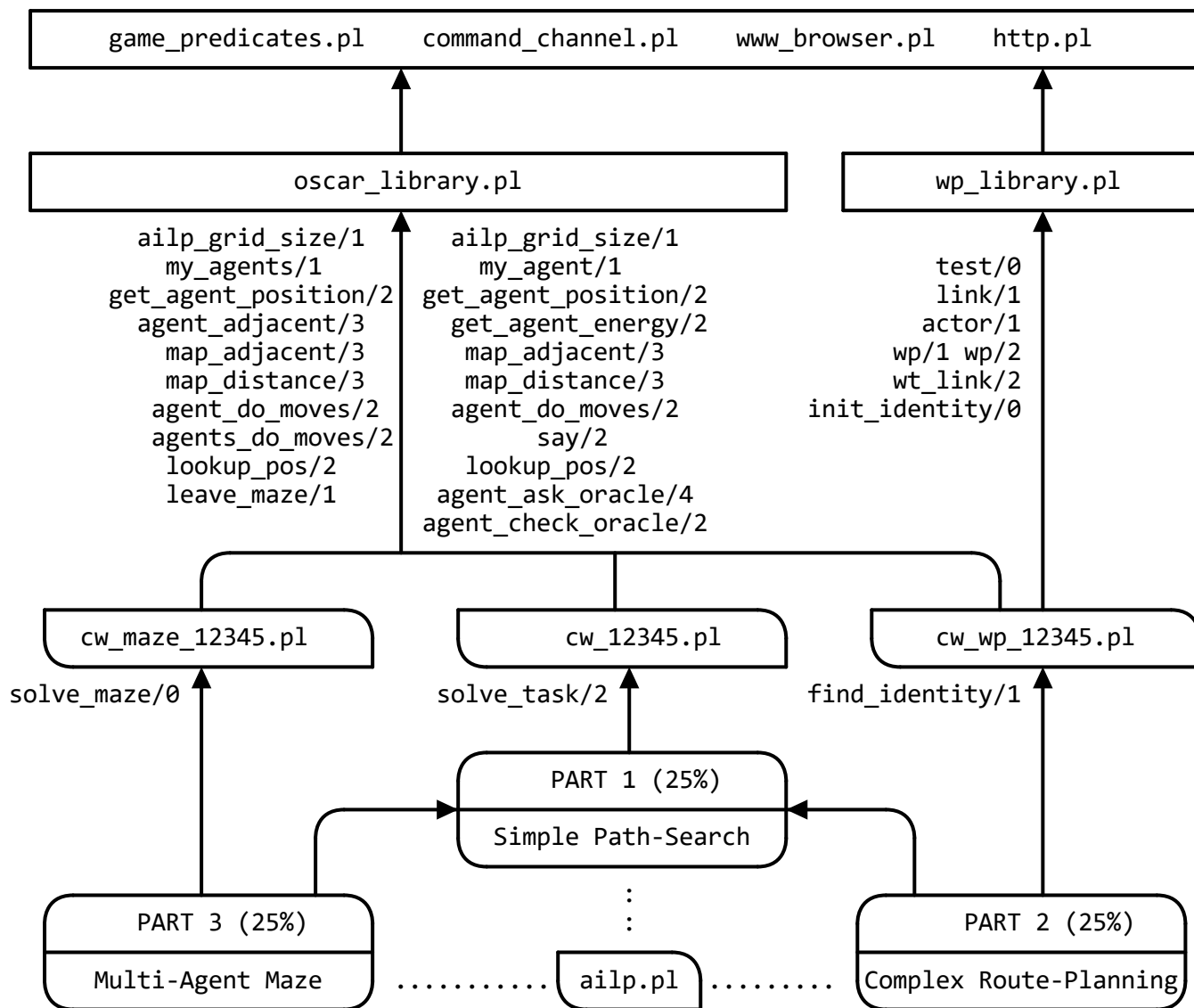
- *plain rectangles* show library modules (with the exported predicates you may use);
- *round rectangles* show parts of the assignment (along with their weighting);
- *semi-round rectangles* show the files (and predicates) you'll need to modify.

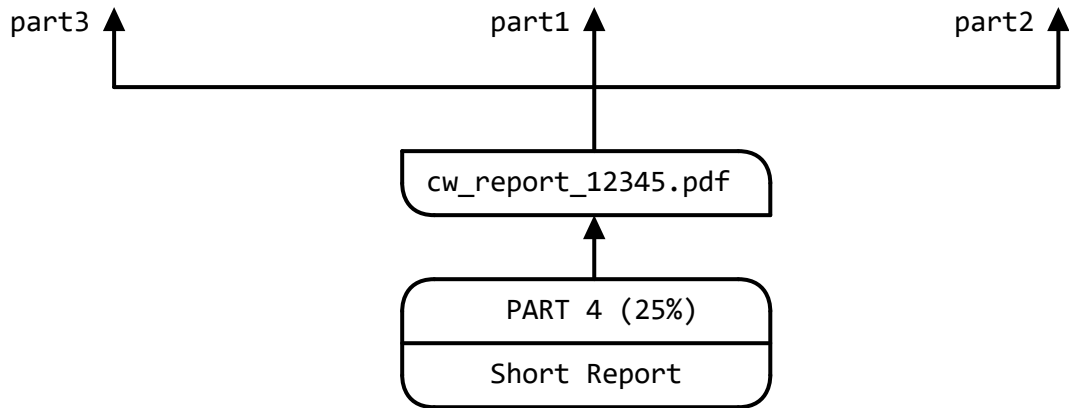


In this coursework, you will not need to look at the **library code** or understand how it works - although you are welcome to look inside the given files if you wish!



Remember that **12345** should be replaced by your (7-digit) **student number**.





**Figure 1:** Overview of part/file/predicate dependencies and grid configurations

The above dependencies are set up by a **file loader** `ailp.pl` that allows you to run the different parts  $n \in \{1, 2, 3\}$  of the assignment from the command line. After moving to the library **root folder** (containing `ailp.pl` file) you should run the following command:

- on a **Linux** machine type `./ailp.pl cw part $n$`  in a **bash** terminal (if necessary, after first making the file executable with the command `chmod +x ailp.pl`)
- on a **Windows** machine type `swipl ailp.pl cw part $n$`  or `swipl-win ailp.pl cw part $n$`  in a **cmd** or **powershell** terminal; or double click on `ailp.pl` in an explorer window and type `cw part $n$`  at the Prolog prompt



When running the above commands, remember to replace the symbol  $n$  by the actual digit (1, 2 or 3) representing the required **part number**:

To enable **interaction with the user**, the GridWorld provides the additional commands, shown below, to open and close a session. The commands in the top table allow the user to set up a local web-server, display the grid in a browser window, add one or more agents to the game, (re-)initialise the grid, and start the game running. This setup must be completed before the GridWorld library predicates can be called; The commands in the bottom table allow the user to remove agents from the game (one-by-one) and stop the web-server. These commands must be executed, at the Prolog prompt, in the order shown below:



Command	Meaning
?-start. .	start web-server
?-join_game(-A).	add a new agent to the game and return its integer ID (e.g. 1) - only 1 agent allowed in Parts 1 and 2
?-reset_game.	(re-)initialise the grid (and stop the game if it is running)
?-start_game.	begin the game



The extra " ." after the "start." command can be optionally used to accept the dialog box which asks the user whether to "open in browser window (y/n)"



Remember to hit "run" in the browser window after starting the web-server (or nothing else will happen until you do)!



Don't forget to use `make` and `reset_game` after you **make changes** to your code (and you may also need to refresh your browser window).

Command	Meaning
?-leave_game.	remove (the current) agent from the game
?-stop.	stop web-server

**Table 2:** *commands to start (top) and end (bottom) a GridWorld session.*

In order to further facilitate user interaction during a session, the GridWorld also provides an **interactive user shell** that can be invoked with the following command that allows the user to run the set of **macros** listed below (which are especially useful for Parts 1 and 2):

Command	Meaning
?-shell.	open interactive shell that provides the following macros



Note that, while you are **not required to use the shell** in this coursework, the following macros can significantly **reduce the amount of typing** required and will also **print some helpful messages** (to the Prolog console and GridWorld window):

Macro	Meaning
?help.	% display a list the macros below
?stop.	% exit from this command shell
?setup.	?-join_game(A),reset_game,start_game.
?reset.	?-reset_game,start_game.
?whoami.	?-my_agent(A)
?position.	?-my_agent(A),get_agent_position(A,P)
?energy.	?-my_agent(A),get_agent_energy(A,Energy).
?topup(+Stat).	?-my_agent(A),agent_topup_energy(A,Stat).
?ask(+Orac,+Qu).	?-my_agent(A),agent_ask_oracle(A,Orac,Qu,Ans).
?call(+G).	?-findall(G,call(G),L).
?search.	?-search_bf % Lab Search
?go(+Pos).	?-solve_task(go(Pos),Cost). % Part 1
?find(+Obj).	?-solve_task(find(Obj),Cost). % Part 1
?demo.	?reset, find(o(1)), ask(o(1),'What is the meaning of life, the universe and everything?'), go(p(7,7)), energy, position, go(p(19,9)), energy, position, call(map_adjacent(p(19,9),_P,_0)), topup(c(3)), energy, go(p(10,10)), energy. % Part 1
?identity.	?-find_identity(ActorName). % Lab Identity, Part 2
?maze.	?-solve_maze. % Part 3

Table 3: Possible shell macros.



Note how the **shell prompt** "?" omits the dash in the standard **Prolog prompt** "?-".



Notice how some of these macros refer to the **user predicates** solve\_task/2, find\_identity/1 and solve\_maze/0 that you'll need to (re)define in **Parts 1, 2 and 3** - so you can use the shell to help you **test** the behaviour of your code.



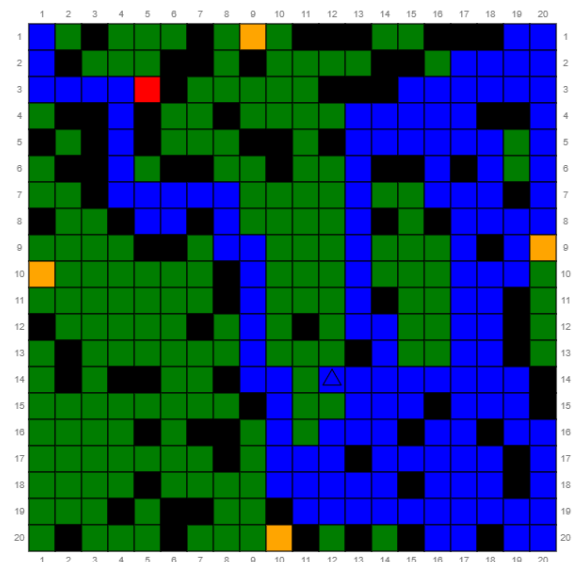
Note that the above **commands** and **macros** should **not themselves be used in any code you submit** as they are only there to help you run and test that code!

The easiest way to get a feel for how the GridWorld works is to run the Part1 **demo** from the library command shell, which will take you step by step through a preprogrammed use case involving the following sequence of main operations:

1. reset the game and execute a path to oracle 1;
2. ask oracle 1 "What is the meaning of life, the universe and everything?";
3. execute a path to position 7,7;
4. execute a path to position 19,9;
5. verify that the agent is adjacent to charge station 3;
6. top up the agent's energy;
7. execute a path to position 10,10 (default code fails!).

As you can see from the following figure, because your agent has a limited supply of energy which gets used up as it moves around and queries oracles, and because it only comes with a naive depth-first search algorithm, by default the agent runs out of energy on this last task - which is an outcome that you'll need to change as you work through this coursework!

```
? demo.
? reset
: reset_game,start_game
! ok
<return> to continue
? find(o(1))
: user:solve_task(find(o(1)),A)
! 5
<return> to continue
? ask(o(1),What is the meaning of life, the universe and everything?)
: my_agent(A),agent_ask_oracle(A,o(1),What is the meaning of life, the universe and everything?,B)
! 42
<return> to continue
? go(p(7,7))
: user:solve_task(go(p(7,7)),A)
! 9
<return> to continue
? energy
: my_agent(A),get_agent_energy(A,B)
! current_energy(76)
<return> to continue
? position
: my_agent(A),get_agent_position(A,B)
! current_position(p(7,7))
<return> to continue
? go(p(19,9))
: user:solve_task(go(p(19,9)),A)
! 64
<return> to continue
? energy
: my_agent(A),get_agent_energy(A,B)
! current_energy(12)
<return> to continue
? position
: my_agent(A),get_agent_position(A,B)
! current_position(p(19,9))
<return> to continue
? call(map_adjacent(p(19,9),_50,_52))
: findall(map_adjacent(p(19,9),A,B),call(map_adjacent(p(19,9),A,B)),C)
! map_adjacent(p(19,9),p(19,10),empty)
! map_adjacent(p(19,9),p(20,9),c(3))
! map_adjacent(p(19,9),p(19,8),empty)
! map_adjacent(p(19,9),p(18,9),t(48))
<return> to continue
? topup(c(3))
: my_agent(A),agent_topup_energy(A,c(3))
<return> to continue
? energy
: my_agent(A),get_agent_energy(A,B)
! current_energy(100)
<return> to continue
? go(p(10,10))
: user:solve_task(go(p(10,10)),A)
! This failed.
<return> to continue
? energy
: my_agent(A),get_agent_energy(A,B)
! current_energy(0)
<return> to continue
```



Terminal output

GridWorld output

**Figure 2:** *Result of running Part 1 Demo with Skeleton Code*

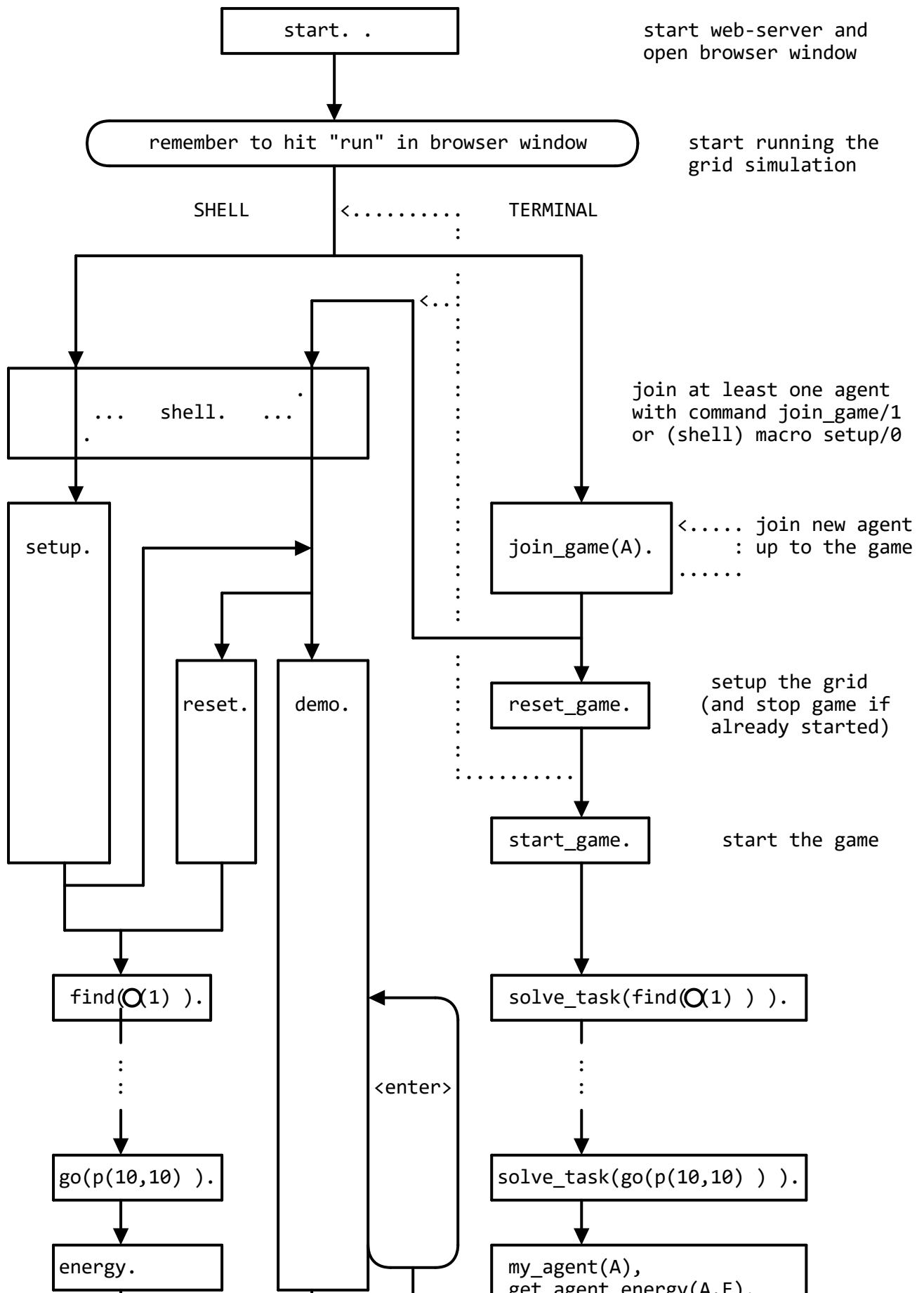
The simplest way to run the demo is with the following sequence of commands which will allow you to easily step through the complete command sequence by just repeatedly pressing the <enter> key after each sub-task completes (or using <ctrl>-c followed by a and <enter> if there is a problem with your code and you need to abort out of the shell):

- ?- start. .
- ?- shell.
- ? setup.
- ? demo.

Various other ways of running the demo are illustrated below - using either the built-in demo macro (centre), or running the constituent shell macros individually (left), or calling their corresponding GridWorld library expansions (right):



The vertical alignment and extent of each box is intended to show the correspondence between the macros and their constituents or expansions:



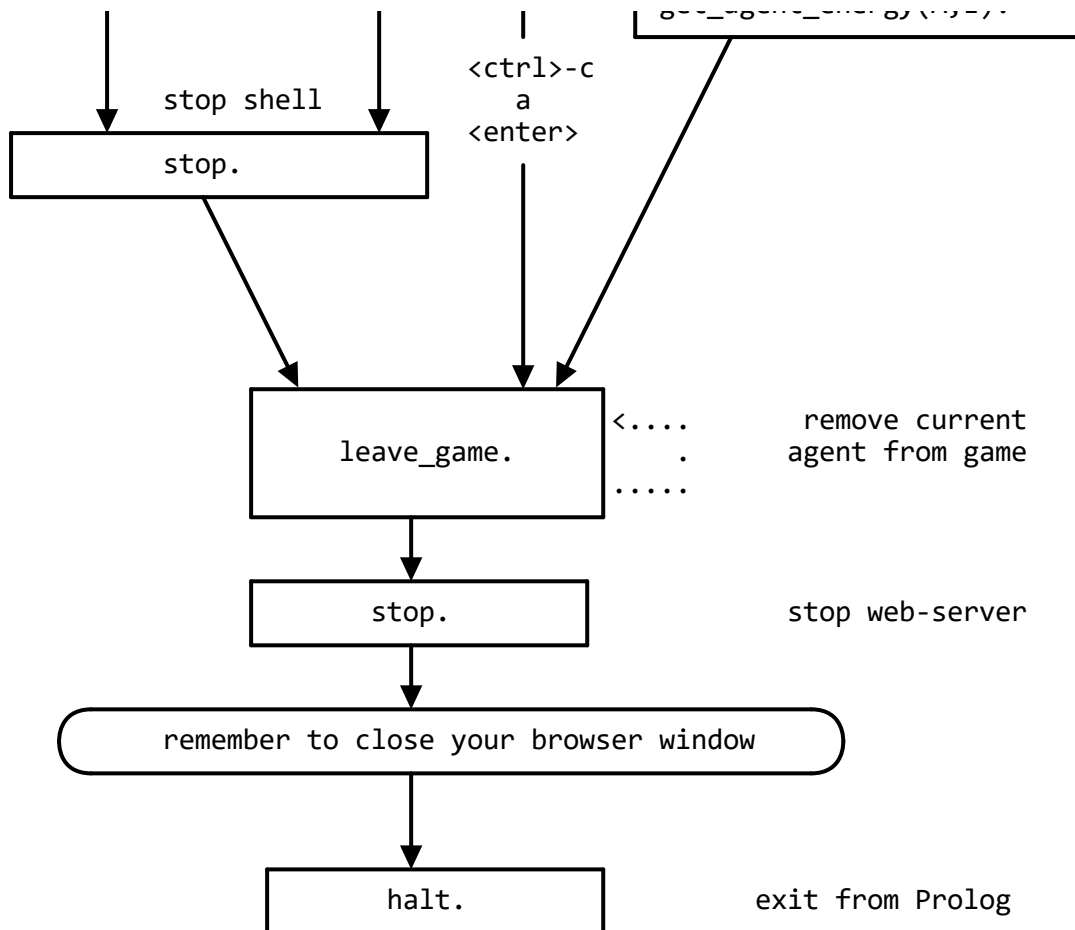


Figure 3: Some different ways the user can run the GridWorld demo



You can enter and leave the user shell at any time; and you can run non-shell commands from within the shell by wrapping them up as an argument to a **call** macro - which will implicitly find *all solutions* of the specified goal. If you want to make your head hurt, try running `call(shell)` from within the shell!



The dotted lines show some ways you can add **more agents** to the game (applicable in **Part 3** which is not restricted to one agent). To do this when the game has started (after a call to `start_game` or one of the macros `setup`, `reset` or `demo`), you'll need to run a precise sequence of commands comprising a `reset_game`, one or more `join_game`, another `reset_game` and a `start_game`. The first `reset_game` is needed to stop the game before more agents can be added (or you'll get an error). To do this from the shell you'll need to explicitly run `call(reset_game)` as the macro `reset` also calls `start_game` which sets the game running again.



Make sure the game is **not paused** in the browser when you call `reset_game` or the server may hang due to feature (or bug) in the way http responses may be sequenced.

Although you **won't need** to exploit this fact in the coursework, the library allows multiple **clients** to interact with the **server** over **http** via an internal predicate `query_world/2` that enables one or more Prolog threads running on your machine to join agents to a game, move them around and query the grid. The only thing you need to know is that, because this all happens over http, calls that involve **looking up the contents of a cell** or **moving an agent** will have a significant **time overhead** as compared to standard Prolog queries.



One of the main ways of **speeding up your code** (and potentially gain marks) will be to **avoid unnecessary interaction** with the grid - such as by executing short paths and not querying the contents of unnecessary cells.



During program development, you are welcome to use SWI built-ins from the [statistics library](#) such as `statistics(+Key,Value)` with the Key `cputime` to run some empirical tests on the **execution time** of various GridWorld queries.



You may also look at **http interactions** in your browser by looking in the **networking tab** of the dev menu for your web browser. This can usually be opened using F12.

An example of the requests made to the web browser at a given timestep is shown below:

```

{commands: [[1, "move", 1, 3], [1, "colour", 1, 3, "blue"], ["god", "redraw"]]}
  commands: [[1, "move", 1, 3], [1, "colour", 1, 3, "blue"], ["god", "redraw"]]
    ▶ 0: [1, "move", 1, 3]
    ▶ 1: [1, "colour", 1, 3, "blue"]
    ▶ 2: ["god", "redraw"]

```

Figure 4: Example http request for agent 1 (blue) to moving to  $p(1,3)$

Parts 1-3 of this coursework all involve you navigating one or more agents around **square grids** of various sizes and configurations in order to solve different tasks (illustrated in Figure 1). In all of these tasks, agents may **move one step** at a time to any empty adjacent (on-grid) cell **immediately** to the **South, East, North** or **West** of the current position (with multiple possibilities being returned in that order). The **location of each cell** in the grid is represented by a Prolog term  $p(X,Y)$  where  $X$  and  $Y$  are natural numbers denoting the horizontal and vertical offsets (rightwards and downwards) from the top left corner. The **contents of each cell** in the grid is represented by exactly one of the following Prolog terms (where  $N$  is an integer identifier of the corresponding object):

Term	Colour	Object	Meaning
$a(N)$	random	agent	a user-controllable agent is located at this position (colour and shape chosen at random)
empty	green	empty space	an agent adjacent to this cell can move here at a cost of $e_{mov} = 1$ energy unit
$c(N)$	orange	charge station	an agent adjacent to this cell can top its energy up to the maximum value $e_{max} = \lceil n^2/4 \rceil$ (which is 100 units on the standard $20 \times 20$ grid)
$o(N)$	red	oracle	an agent adjacent to this cell can ask the oracle (at most) one question at a cost of $e_{ask} = \lceil e_{max}/10 \rceil$ (which is 10 units on the standard $20 \times 20$ grid)
$t(N)$	black	thing	these represent "walls" or "obstacles" that your agent cannot move through
unknown	grey	hidden	an agent will need to visit an adjacent cell to reveal the contents of this cell (in Part 3)



Note that  $\lceil . \rceil$  represents the Prolog **ceiling** function which can be used within an **is** to round its argument up to the nearest integer (i.e. towards positive infinity).



Although the GridWorld library supports the random movement of walls, oracles and charge stations, Parts 1-3 will all be restricted to **static grids** where only agents will be allowed to move. Moreover, in order to help you test your code and get a realistic estimate of its performance, Parts 1-3 will all impose strict restrictions on the number of various objects in grid and will ask you to define specific predicates to solve different tasks, as detailed in the following sections. Of course, you are welcome to consider alternative static and dynamic variations of these task definitions and grid configurations in Part 4.

The marking of Parts 1-3 will be mainly carried out by an **auto-tester** that will take into account several **marking criteria** specified in the following sections. For each criterion, you will gain from 1 mark for a **basic solution** (that provides some slight improvement over any baseline code) up to 5 marks for (near-) **optimal solutions**. In cases where there might be a potential **trade-off** between two or more of the objectives, their respective **priorities** will be stated in order of importance (**primary**, **secondary**, etc); and your mark for any higher priority criteria will serve as an **upper bound** on your mark for any lower priority criteria (as well helping to adjust the **expected performance**).

To avoid losing marks, you should take care not to **over-optimize** low priorities at the expense of high priorities. Before submission, you should also **double check** that any **auto-tested predicate** definition is in the correct **file** with the correct **name** and **arity**. While you are free to write any **helper predicates** you like (with names and arities of your choosing) you should ensure all of your code only make calls standard **SWI built-ins** or explicitly exported **library predicates** (as any attempt to modify or expose other library definitions won't work on the auto-marker).

As **good coding practice** is part of the marking scheme you are strongly advised to: **comment your code** to explain the meaning of each argument and the behaviour of each predicate; **format your code** to make the logical flow clear (using **informative variable and predicate names**); **test your code** to make sure it compiles **without errors** and that predicates **terminate properly** which means that wherever possible they should be **(semi-)deterministic** in the sense that they should terminate after succeeding once (or failing). You should especially try to avoid **non-termination** and **run-time exceptions** or predicates that return **duplicate solutions** or leave unnecessary **choice points**.

Note that you won't get any marks for solutions that don't compile or are half finished. You will get marks for improving on the default working skeleton code you have been given.

## 4 Part 1

---

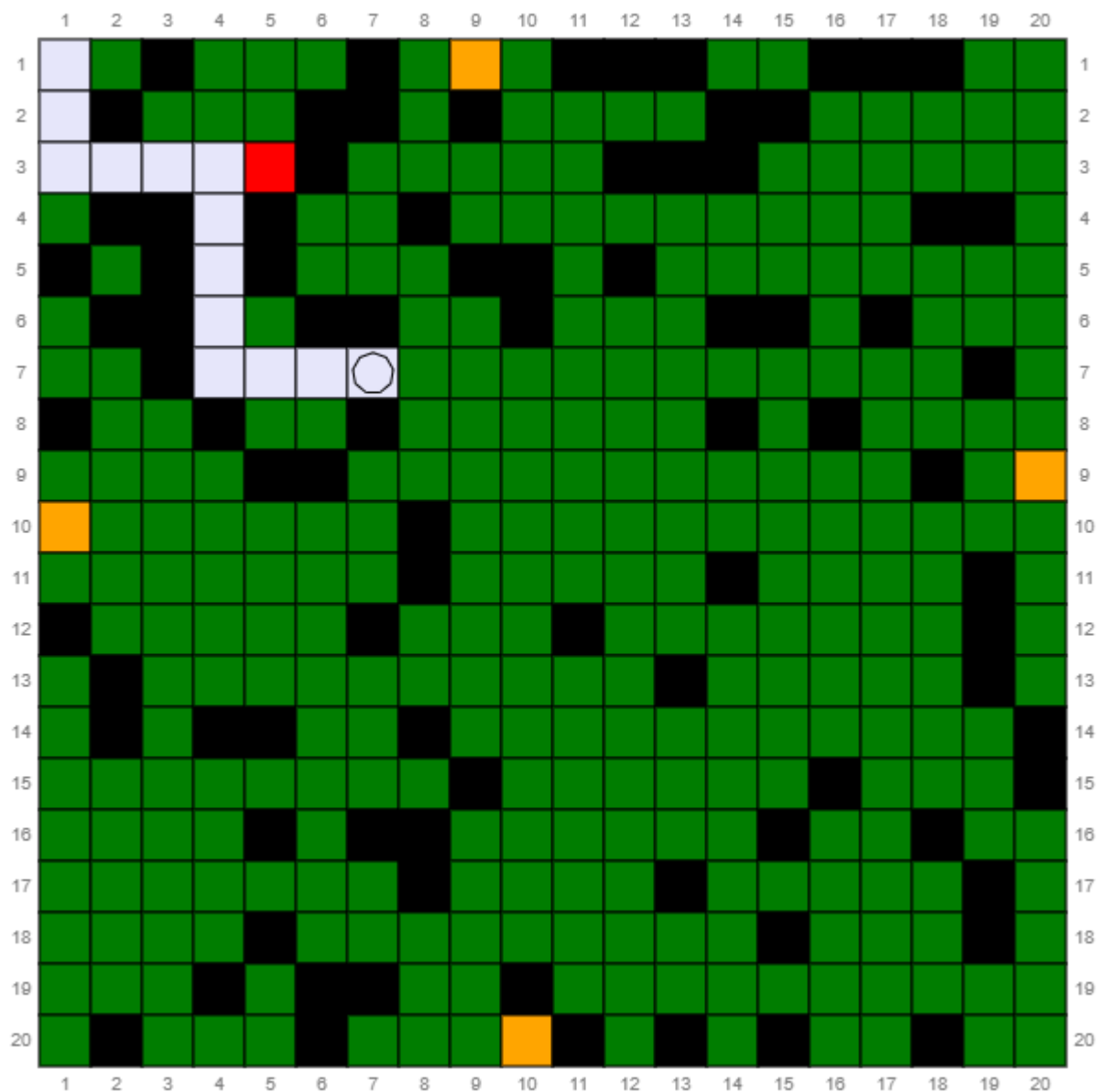


Run Part 1 using `./ailp.pl cw part1` (Linux) or `swipl ailp.pl cw part1` (Windows)

Part 1 of the coursework (worth 25%) requires you to write **path finding** code that allows a **single agent** on a **square grid** to efficiently navigate to a specified **cell** or **object** (charging station or oracle) without running out of energy.

To provide you with a stable environment for debugging, the Part 1 library will always set up the same **demo configuration** depicted below for 1 agent (white) on a 20×20 grid (green) with 4 charging stations (orange), 1 oracle (red) and 84 obstacles (black); where the agent always starts in the top left corner at position  $p(1,1)$  with a 'full tank' of 100 unit of energy.

Each **move** will cost **1** unit of energy, and each oracle the agent visits can be **queried**, at most once, at a cost of **10** units of energy. But the agent's energy can be **topped up** back to **maximum value** of **100** by visiting any of the charging stations (any number of times).



*Standard grid configuration for Part 1: after agent has visited  $o(1)$  and moved to  $p(7,7)$*



If your agent runs out of energy when it is not adjacent to a charging station, then no further moves are possible and your only option will be to reset the game.

The point of Part 1 is to write code that enables your agent to (efficiently) find a (short) path from its current location to a specified cell or object. More specifically, you will need to write a predicate `solve_task(+Task, -Cost)` that solves Tasks of the form shown below:

Task	Meaning
<code>go(?Pos)</code>	execute a path to a grid position that unifies with <code>Pos</code> (which, if set to <code>p(4,5)</code> or <code>p(X,6)</code> , would leave the agent at position 4,5 or the nearest accessible cell in row 6, respectively)
<code>find(?Obj)</code>	execute a path to a grid position adjacent to an object that unifies with <code>Obj</code> (which, if set to <code>c(1)</code> or <code>o(X)</code> , should leave the agent next to charge station 1 or the nearest accessible oracle, respectively)

The predicate `solve_task/2` should either succeed after executing a path to the target (and returning the cost of the path - defined as the number of moves made) or it should simply fail if the task is not feasible (because obstacles prevent it from reaching the goal or because it would run out of energy in the way). Your solution should make use of the following library predicates (some of which should be familiar from *Lab Search*) to interact with the grid:

Predicate	Meaning
<code>my_agent(-A)</code>	returns integer ID of the first Agent which joined to (but was not subsequently removed from) the game
<code>map_adjacent(+Pos,-Adj,-Obj)</code>	given a grid Position, return any Adjacent on-grid cell location along with the Object it contains (or the term <code>empty</code> )
<code>map_distance(+Pos1,+Pos2,-Dist)</code>	given two cell locations $Pos1=p(x1,y1)$ and $Pos2=p(x2,y2)$ , returns Manhattan (taxi-cab) Distance defined as $Dist =  x1-x2  +  y1-y2 $
<code>lookup_pos(+Pos,-Obj)</code>	return Object located at given Position (n.b. <b>cannot</b> be used the other way around to return Position of a given Object!)
<code>get_agent_energy(+A,-Energy)</code>	return Agent's current Energy
<code>get_agent_position(+A,-Pos)</code>	return Agent's current Position
<code>agent_topup_energy(+A,+Station)</code>	top up Agent's energy at charge Station
<code>agent_do_moves(+A,+Path)</code>	execute sequence of moves in given Path to Agent on grid (or fail at the first point where a move is found to be invalid)
<code>say(+Message,+A)</code>	print Message next to the current Agent on the grid (maybe useful for debugging?)

**Table 4:** *Library Predicates for Part 1.*



Note that `lookup_pos/2` with modes `lookup_pos(+Pos,-Obj)` must have an instantiated first argument when you call it. So, given a Position, it returns an Object, not vice versa.



Although you, the user, will be able to see the layout of the grid in the browser window, please remember your agent can only obtain that information by making calls to the relevant cells using `map_adjacent/3` or `lookup_pos/2` (which are comparatively expensive as they operate over http). Thus, the tasks of efficiently finding an optimal path to a given location or finding the location of a given object are non-trivial (given the restrictions imposed by the above mode declarations):

Your skeleton submission file `cw_12345.pl` contains an initial version of `solve_task/2` which finds (poor) solutions with a (naive) backtracking depth-first search `solve_task_dfs/3` that uses the predicate `achieved/2` to detect goal states, as defined below:

```
solve_task(Task, Cost) :-
    my_agent(A), get_agent_position(A, P),
    solve_task_dfs(Task, [P], [P|Path]), !,
    agent_do_moves(A, Path), length(Path, Cost).

solve_task_dfs(Task, [P|Ps], Path) :-
    achieved(Task, P), reverse([P|Ps], Path)
    ;
    map_adjacent(P, Q, empty), \+ member(Q, Ps),
    solve_task_dfs(Task, [Q, P|Ps], Path).

achieved(Task, Pos) :-
    Task=find(Obj), map_adjacent(Pos, _, Obj)
    ;
    Task=go(Pos).
```

You will need to replace the above `solve_task_dfs/3` by a new predicate (with your choice of name and arity) that implements an appropriate A\* search strategy. The easiest way to do this is by the following three consecutive steps:

- First substitute `solve_task_dfs/3` by a **breadth-first** search predicate which can be easily adapted from the `search_bf/2` of *Lab Search* by simply: adding an extra argument to store the Task; and calling `achieved/2` instead of `complete/1`.
- Then convert your predicate into an **A\* search** that sorts the agenda using a score obtained by summing the distance of a node from the start with its Manhattan distance to the target (if its location is known) or with the constant 0 (if it's not).
- Finally you'll need to adapt your code to take into account the agent's available energy and also gives it the ability to **refuel** at charging stations along the way, if necessary.



Recall that the scoring function  $f = g + h$  used to order the nodes in an A\* agenda is the sum of the actual cost  $g$  from the start position to the current node and an (optimistic) heuristic estimate  $h$  of the cost from here to a goal. Thus, if the location of the **target is known** in advance (as in a ground `go` task), then setting  $h$  to the Manhattan distance will result in A\* finding optimal paths (around any intervening walls) while querying relatively few cells on average. But, if the location of the **target is unknown** (as in a `find` task), then setting  $h = 0$  will revert A\* back to a plain breadth-first search, which is about as good as one can do, on average, in random grids (for the reasons in the next box).





Note that any **a-priori** attempt to find the location of an object (by grid search or random sampling) in order to use directed (Manhattan) A\* for more efficient path finding will most likely result in querying **more cells** than needed by blind (breadth-first) A\* to return an optimal path in the first place! This is because breadth-first search is guaranteed to only query cells **reachable** by the agent (as opposed to disconnected islands - such as cells 14 and 15 in the first row of the standard grid) and it can also be used to non-deterministically return paths to objects in order of closeness) when solving queries like `solve_task(find(c(X)))`.



You may use the built-ins `sort/2` and `ord_union/3` for sorting a list and merging two sorted lists, respectively. You may also use the built-ins `var/1` and `ground/1` to check if a term is a variable or if it is variable-free.



If there is no (in)direct route from the current position to the target using the current energy, then `solve_task/2` should **fail** (without moving at all) - thereby avoiding the situation in the demo where the skeleton code runs out of fuel.



In Part 1, you **should not** worry about the agent's energy level at the goal. You should simply execute a shortest possible path to the goal - even if it means the agent wouldn't have enough energy to **subsequently** reach another charging station in order to top itself up again.

Your Part 1 code will be tested on the standard **demo grid** (generated by the Part 1 library code) along with a number of random **test grids** (generated by the Part 2 library code - in which the grid size and the number or location of objects may differ slightly) and a suite of hand-crafted **edge cases** (subject to the same restrictions stated in Part 2, below). Your code will be marked according to the following 5 criteria (relative to the baseline skeleton code):

- up to **5 marks** for minimising the number of **moves made**<sup>†</sup> when executing direct paths in the standard and random grids;
- up to **5 marks** for minimising the amount of **time taken**<sup>††</sup> when finding direct paths in the standard and random grids;
- up to **5 marks** for also being able to find and execute **indirect paths** that allow agents to reach distant targets by refuelling at one or more charge stations en-route;
- up to **5 marks** for correctly handling some tricky or unusual **edge cases** that will be set up on specially hand-crafted grids<sup>§^{\dagger\dagger\dagger}\dagger\dagger\dagger§</sup>;
- up to **5 marks** for demonstrating **good coding practice**.



<sup>†</sup> As the number of **moves made** will be minimised by both breadth-first and A\* search, you will gain *all* of the marks for a *correct* implementation of either of these

(or an equivalent); And you should be able to get a good idea if your paths are optimal by simply running the standard demo.



†† As the **time-taken** will be dominated by calls to `agent_do_moves` and `map_adjacent` (which have a significant http overhead), You will gain *most* of the marks for a *correct* implementation of A\* search - which will minimise the time of `agent_do_moves` by returning optimal paths and, assuming no major coding inefficiencies (like adding duplicate nodes to the agenda), will also reduce the number of `map_adjacent` calls on average. You should be able to verify this by ensuring each task in the standard demo finishes in a few seconds (as opposed to minutes or hours). While further optimisation is possible (e.g. by remembering the contents of previously seen cells) the gains will be diminishing and small tweaks to save a few hundred milliseconds won't be worth the effort.



††† You will be given **no additional information** about possible edge cases; it is YOUR job to think about what they might and deal with them appropriately!



As well as testing your code on the standard Part 1 demo (above), you are advised to also run your code on the random grids of Part 2 (see below) to get a better idea of your code performs in wider variety of configurations. Note that your Part 1 code will automatically be imported when you invoke Part 2 and you are welcome to run the standard demo - even though some of the tasks may now fail (as they become edge cases) in the random grid configurations.



Although it is not required in this coursework, you are encouraged to consider investing some time to develop a mechanism for exporting and importing specific grid configurations so that you can more easily develop you own test cases and easily test your code on them. But remember that any changes you make to library files will not be carried over to the automarker - so, before submitting, please remember to make sure your code works with the original library code.

## 5 Part 2

---



Run Part 2 using `./ailp.pl cw part2` (Linux) or `swipl ailp.pl cw part2` (Windows)

Part 2 of the coursework (worth 25%) involves you writing **route planning** code for a **single agent** on randomly generated **square grids** of integer size  $n$  between 15-25 with 2-4 **charging stations**, 1-10 **oracles** and about 80-100 **obstacles**. The cost of a move is still  $e_{mov} = 1$  unit of energy; the agent's energy can still be **topped up** to its maximum by visiting

any charging stations (any number of times); and each oracle the agent visits can still be queried at most one. But, now, the maximum energy is defined more generally as  $e_{max} = \lceil n^2/4 \rceil$  (which is still 100 on a 20x20 grid); the cost of querying an oracle is defined more generally as  $e_{ask} = \lceil e_{max}/10 \rceil$  (which is still 10 on a 20x20 grid); and the agent will start at a random position with a random energy that is between 50-100% of  $e_{max}$ .



You should ensure your code works out the appropriate starting values and costs in both Part 2 and Part 1 (and does not rely on hard-coded integers for these).

To provide you with a varied environment for debugging, the Part 2 library will always set up a different **random configuration** like the one shown below for 1 agent (pink) on a 20x20 grid (green) with 2 charging stations (orange), 10 oracles (red) and 81 obstacles (black). In these scenarios, the agent will start with random energy at a random position on a grid with size between 15 and 25. In the figure below it started at 5,12 and visited 4 oracles.



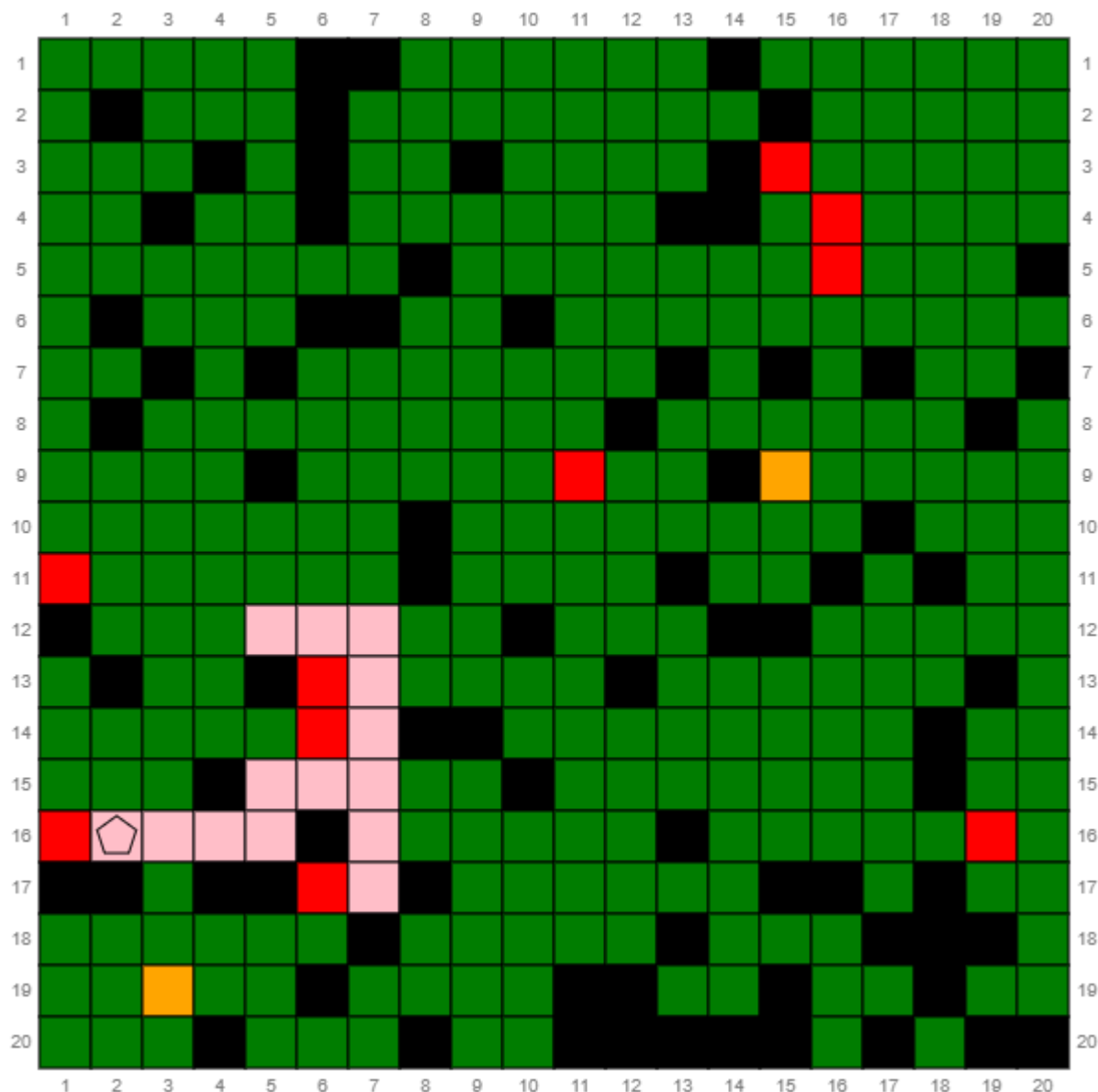


Figure 5: A representative grid configuration for Part 2

The point of Part 2 is write code that allows your agent to plan complex routes that visit as many oracles as possible via as many charge stations as necessary in order to maximise your chances of finding a secret actor identity by asking each oracle to return a random link from that actor's Wikipedia page. And this part also indirectly illustrates how Prolog can be used to perform http requests and parse html files.

Please note the following:

- `find_identity/1` will need to be defined in `cw_wp_12345.pl` not in `cw_12345.pl` or `lab_identity_12345.pl`
- This part provides predicates for connecting via http to live Wikipedia pages and extracting WikiLinks from the WikiText contained in those pages;

- Therefore, you will need to have a live internet connection and you may need to grant appropriate firewall permissions to the GridWorld Webserver;
- You will need to join to the game a single regular agent (whose id is returned when you first called `join_game(A)` or when you later call `my_agent(A)`);
- You will have to physically **move** your agent next to an oracle (which may not always be possible due to obstacles or energy constraints) in order to ask it for a link;
- There are now **several oracles** (which makes path-finding more complex)
- Your agent will only be able to consult each oracle at most once;
- Each time you query an oracle, it will cost some energy (one tenth of the maximum);
- As oracles are independent, they may **repeat** a link (answer) that you had already previously been given by another oracle;
- Thus, sometimes, you may **fail** to determine your identity even after visiting the maximum possible number of oracles.
- The randomness of the grid may mean some oracles are not accessible.
- To simplify this task, your secret actor identity will be chosen from a set of 12 pre-defined famous Hollywood actors, and oracles will only return links from a set of 15 predefined links drawn from those actors' Wikipedia pages (which will greatly reduce the number of Wikipedia pages you need to process)!

You will need to write a predicate `find_identity(-ActorName)` that executes a path to as many oracles as possible and either returns the secret actor's name, if it can be determined, or returns the term `unknown` if it cannot. In addition to earlier Part 1 predicates in [Table 4](#), you will need the following additional library predicates for Part 2 :

Predicate	Meaning
<code>actor(-A)</code>	Bind A to one of the 12 possible actors
<code>link(-L)</code>	Bind L to one of the 15 possible links
<code>wp(+Q)</code>	Retrieve the text from the Wikipedia page titled Q and print to stdout
<code>wp(+Q,-WT)</code>	As above, but instead of printing it will bind the text to WT
<code>wt_link(+WT,-Link)</code>	Bind Link to a link inside the supplied WikiText
<code>init_identity</code>	Randomly select a new identity, this should only be used for testing purposes
<code>test</code>	Test that <code>find_identity(A)</code> succeeds for each possible actor <sup>†</sup>
<code>agent_ask_oracle(+A,+Orac,+Qu,-Ans)</code>	return Oracle's Answer to Agent's Question (requires Agent to be adjacent to Oracle and costs 1/10 of max energy). If Qu=link then the oracle will return a random link from the secret actor's Wikipedia page
<code>agent_check_oracle(+A,+Orac)</code>	check if Agent has already queried this Oracle

**Table 5:** *Additional Library Predicates for Part 2.*



<sup>†</sup> Note that, `test` may only succeed in finding the identity of a few actors, as your agent can query each oracle at most one once!

The **primary aim** of Part 2 is to maximise the number of oracles that you visit (as that gives you the best chance of finding your identity); The **secondary aim** is to minimise the number of steps in the solution; and the **tertiary aim** is to minimise the execution time.

Your skeleton submission file `cw_wp_12345.pl` provides the following initial version of `find_identity/1` which (sometimes) finds (poor) solutions by (naively) using `solve_task/2` from Part 1 to execute paths to consecutive oracles in turn (without considering how much fuel is available):

```

actor_has_link(L,A) :-
    actor(A), wp(A,WT), wt_link(WT,L).

eliminate(As,A,K) :-
    As=[A], !
    ;
    solve_task(find(o(K)),_), !,
    my_agent(N),
    agent_ask_oracle(N,o(K),link,L),
    include(actor_has_link(L),As,ViableAs),
    K1 is K+1,
    eliminate(ViableAs,A,K1).

find_identity(A) :-
    findall(A,actor(A),As), eliminate(As,A,1).

```

You are advised to begin by testing out the Wikipedia predicates listed above (to ensure your internet connection and firewall settings are correctly set up). For example, run the query `findall(A,actor(A),As)` to return the list of possible actors; and run the query `wp('George Clooney')` to print the WikiText from this actor's page. Make sure you understand how the skeleton code works by progressively using the built-in `include/3` to whittle down a sublist of viable actors (whose pages contain all the links provided by the oracles consulted thus far) until only one actor is left in that list.

To get a feel for the problem you could initially try to implement either a **brute force** method that tries to enumerate all possible valid paths (of which there are many as you may need to visit the same charging station more than once) or a **greedy approach** that aims towards the closest unvisited oracle (or heads to a charging station whenever the energy goes below some threshold).

By thinking carefully about potential edge cases, you should be able to see that a brute force approach is very expensive, while a greedy approach will often lead to sub-optimal solutions (both in terms of path length and the number of oracles visited). Therefore, you need to decide how much effort you are willing to invest in developing a more intelligent solution (as you will most likely start to experience **diminishing returns** and a **potential trade-off** between the marking criteria).

To speed up your code, you may wish to keep an internal memory of **previously discovered** oracles and chargers so you may quickly direct your agent to a nearby charging station or (unvisited) oracle. You can also consider various ways of keeping track of the energy such as looking for a charger whenever it goes below some threshold (that you could consider empirically testing to find a good value). More intelligent approaches are obviously possible, but will take longer to develop and test.

Your mark for Part 2 will be determined by testing your code on a series of random grids generated by the library itself (which means you can get a good feel of how well your code is doing) along with a suite of hand-crafted edge cases (all of which satisfy the general conditions stated at the beginning of this section).

- up to **5 marks** for the **primary** aim of **maximising the number of oracles** visited;
- up to **5 marks** for the **secondary** aim of **minimising the number of moves** made;
- up to **5 marks** for the **tertiary** aim of **minimising the time** taken;
- up to **5 marks** for correctly handling **edge cases** on hand-crafted grids;
- up to **5 marks** for demonstrating **good coding practice**.

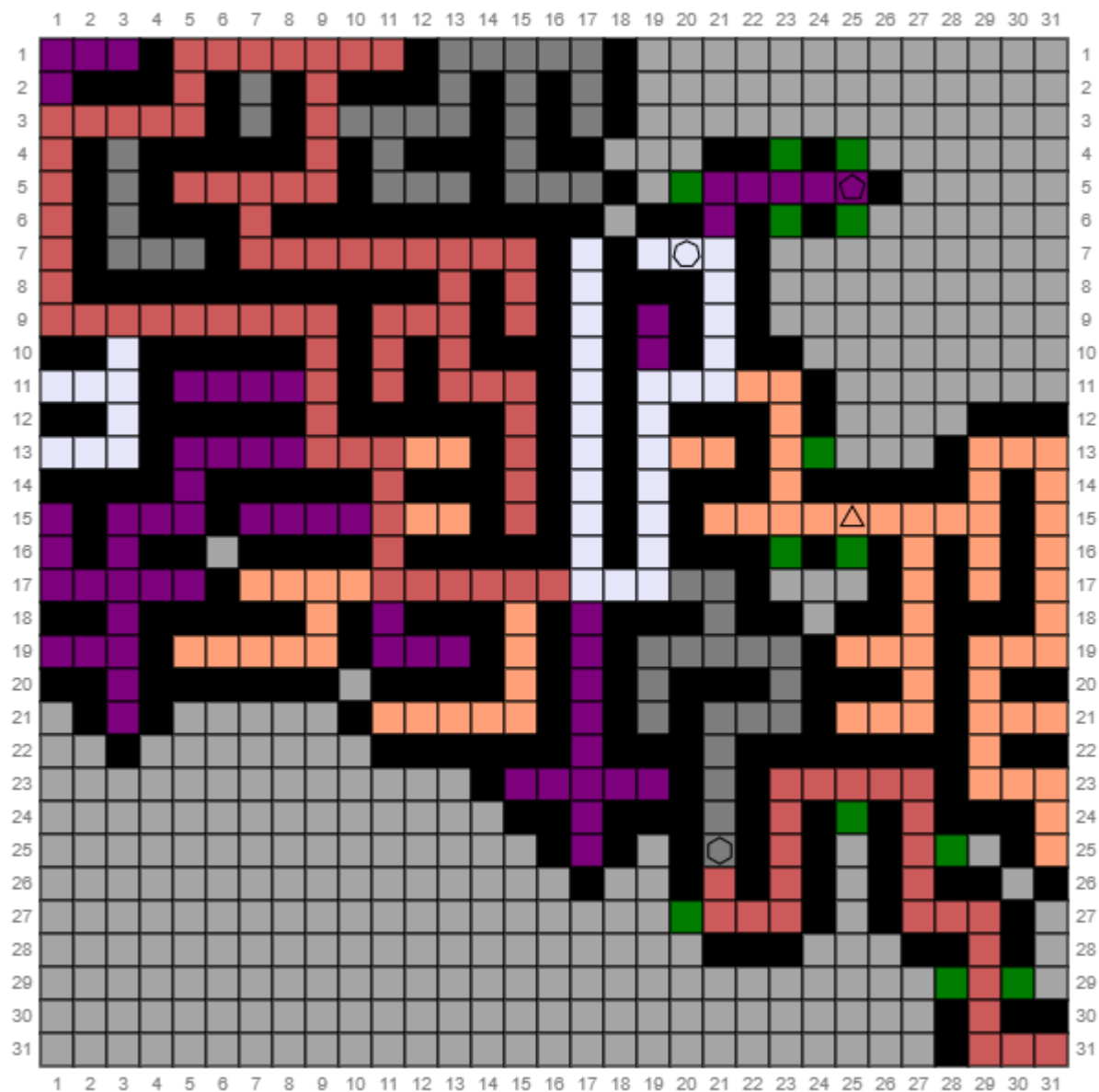
## 6 Part 3

---



Run Part 3 using `./ailp.pl cw part3` (Linux) or `swipl ailp.pl cw part3` (Windows)

In part 3, the grid configuration and the task you need to solve is **very different** from Parts 1 and 2. The game is now played with **1-10 agents** on square grids whose size is given by an **odd number  $n$**  between **11-31** and which is populated by empty cells and walls (with no oracles or charging stations). Each grid represents a **maze** in which there is exactly one path between any two (empty) cells and the width of every such path is exactly one cell wide. The goal is to navigate each of your agents from a **starting location** near the **top left** corner to an **exit position** in the bottom right corner (i.e. at 31,31 in the following figure) where it must call the `leave_maze(+A)` predicate:



*Example maze configuration for Part 3*

Agents can still **move** to an immediately **adjacent** cell (**South, East, West** or **North**) but there is no longer any cost associated with making a move; and there are no oracles or charging stations to visit - so you don't need to worry energy in this part! However, the situation is complicated by the fact that the contents of **each cell is now initially hidden** (denoted by the Prolog term `unknown` and represented by a **grey** square in the browser) **until some agent is initially placed or subsequently moved onto an adjacent cell**. Thus, the only way your agents can find a path through the maze is by physically moving through it to see what is there.



Once an agent has **unhidden** the contents of a cell (by virtue of being located on an adjacent cell), then **each and every** agent in the game will **henceforth** be able to query the contents of that cell.



Because (most of) the maze is initially hidden, any code you wrote for Part 1 (and Part 2) may not work directly in Part 3 as `map_adjacent/3` and `lookup_pos/2` may now return `'unknown'`.



Note that as soon as one of your agents has found the exit, no further exploration of the maze is necessary due to the tree property, mentioned above, that there is only one path between any two points in the maze.

This means there will be much more emphasis on moving multiple agents a single step at a time than on moving a single agent multiple steps. For this reason, Part 3 provides the predicate `agents_do_moves/2` so that **many agents** can **each move** up to **one step** in the **same tick** on the server. For example, `agents_do_moves([1,2],[p(1,1),p(2,2)])`. would move `a(1)` to `p(1,1)` and `a(2)` to `p(2,2)`. In general, this predicate behaves *as if* the moves of the respective agents had been carried out, from head to tail, in the order they are presented in the list. But, instead of executing the moves on successive hypothetical ticks, they are all batched into one actual tick (potentially allowing much faster movement). An example of a such batched server request is shown below. If any move fails, the offending agent will stay in place and any remaining moves will still be attempted.

```

{,--}
  commands: [[4, "move", 1, 2], [4, "colour", 1, 2, "gray"],
    ▶ 0: [4, "move", 1, 2]
    ▶ 1: [4, "colour", 1, 2, "gray"]
    ▶ 2: [9, "move", 5, 6]
    ▶ 3: [9, "colour", 5, 6, "lavender"]
    ▶ 4: ["dyna", "colour", 5, 7, "green"]
    ▶ 5: [5, "move", 6, 5]
    ▶ 6: [5, "colour", 6, 5, "lightsalmon"]
    ▶ 7: [6, "move", 1, 7]
    ▶ 8: [6, "colour", 1, 7, "beige"]
    ▶ 9: ["god", "redraw"]
  ]

```

Figure 6: Example batched request moving 4 agents in a single timestep



Please note that when agents are located close together, the ordering of the lists in `agents_do_moves` may be important - e.g. if one agent must be moved before another can take space it just vacated.

The Part 3 library code also allows you add an extra argument to `agent_join_game` so you can easily specify the number of agents you want to add; and it allows you to get a list of their IDs using `get_agents`. It also makes available a new predicate `agent_adjacent/3` that behaves exactly like `map_adjacent`, but given an agent as input as opposed to a position. Finally, there is the new predicate `agent_leave_maze` which allows the an agent to leave the

maze (if it is located in the bottom right corner). A summary of the available library predicates is given below:

Queries	Meaning
<code>agents_do_moves(+As,+Moves)</code>	simultaneously make each Agent in Agents perform the corresponding Move in Moves.
<code>agent_do_moves(+A,+Moves)</code>	As before, although due to the world being concealed this may be less useful
<code>map_adjacent(+Pos,-AdjPos,-Obj)</code>	This predicate is still imported, although it is slightly less useful with the presence of unknown cells
<code>agent_adjacent(+Agent,-AdjPos,-Obj)</code>	similar to <code>map_adjacent</code> , but takes in an agent rather than a position
<code>lookup_pos(+Pos,-Obj)</code>	Return the object at Pos, if known
<code>my_agent(-A)</code>	This predicate is still included and is true if A is the first agent stored internally. It is best to avoid using this predicate for this section
<code>my_agents(-As)</code>	Binds A to a List of all agents in the current terminal
<code>join_game(+As,+N)</code>	Join N Agents to the game and store the ids in As
<code>get_agent_position(+A,-P)</code>	As before
<code>leave_maze(+A)</code>	Succeeds if A is at the bottom right corner, and makes A leave the game
<code>ailp_grid_size(N)</code>	Bind N to the height/width of the grid



The maze generator will always place one agent in the top left corner `p(1,1)` and will then place any subsequent agents next to a previously placed one. Each agent will be randomly assigned a unique id from 1-10.



A query that attempts to add more than 10 agents to the game will NEVER succeed.

1. To run the assignment execute: `./ailp.pl cw part3`.



```

start. % Then press run in the web browser
join_game(As,1).
% if you want more than one agent then change 1 to any number up to 10
reset_game.
start_game. % after this no further agents can join, until the game is reset

```

2. Implement the predicate `solve_maze/0`, such that it controls all the agents in the game and leads them to the exit of the maze, then calls `leave_maze`

The skeleton code provided in `cw_maze_12345.pl` uses a helper function `find_moves` to identify moves for each agent at a given timestep. It is possible to extend this predicate to complete this task although it is not required. The provided skeleton code does not check that moves are actually possible nor does it leverage any information about the world, so there is a lot of room for improvement.

```

solve_maze :-
    my_agents(Agents),
    find_moves(Agents,Moves),
    agents_do_moves(Agents,Moves),
    solve_maze.

%%%%%%%%%%%%%% USEFUL PREDICATES %%%%%%%%%%%%%%
find_moves([],[]).
find_moves([A|As],[M|Moves]) :-
    findall(P,agent_adjacent(A,P,_),PosMoves),
    random_member(M,PosMoves),
    find_moves(As,Moves).

```



Even though there is no explicit agent to agent communication, it is possible to communicate implicitly through arguments passed into predicates. For instance, it is possible to keep track of visited positions by adding a new argument to `find_moves`. To see a simple example of this, look back to how movement direction is communicated between steps of spiral in **Lab 2**.



Feel free to modify the grid size for testing by changing line 66 of `game_predicates.pl` in order to shorten the time of individual runs. **The grid size MUST be odd or else it will never successfully generate.**

Marks will be awarded as follows:

- **5 marks** will be awarded for a solution which allows a single agent to successfully leave the maze making reasonable moves. This means that the agent will not

repeatedly explore routes which it has determined lead to a dead end and it will successfully leave the maze once it gets to the end.

- **5 marks** will be awarded for a solution that supports 3 agents and makes effective use of them. This means agents will explore different paths and will ideally avoid dead ends discovered by each other. 3 agents forming a conga line and then exploring as though they were one would not be worth any marks for this section.
- **5 marks** will be awarded for everything up to this point working with a variable number of agents, that can range from 1 to 10.
- **5 marks** will be awarded if upon discovery of the end point by one agent, all the other agents navigate to it using near shortest paths.
- **5 marks** will be awarded for efficiency of solutions in terms of runtime.

## 7 Part 4

---

In Part 4, you are required to write a short report (maximum 1000 words) considering how additional AI techniques (beyond those employed in the coursework so far) could be used to extend or improve the performance of the agents that you have coded in Parts 1, 2, and 3, in order to tackle real-world analogues of the kind of search problem that you have considered in the coursework so far.

For this part of the coursework, you are **not** required to code or implement any of these AI techniques or systems and you will **not** gain marks for including any code, or results from running any code, as part of your report. However, you may, if you think it improves your report, include pseudocode to help explain an AI algorithm, technique or system that you are discussing.

The report could be imagined to be a technical briefing written for an engineering audience that is naive with respect to AI in general, but has seen how AI algorithms can be employed to solve the artificial problems dealt with in Parts 1, 2, and 3, and is interested in how AI techniques could be employed to control autonomous agents to complete real-world search tasks.

Marks will be allocated as follows:

- **5 marks** will be allocated for consideration of the **strengths and shortcomings** of the approaches used so far in Parts 1, 2, and 3. i.e., these marks are awarded based on your report's discussion of the extent to which the kind of algorithms developed during Parts 1, 2, and 3, may or may not generalise successfully to real-world search problems.
- **10 marks** will be allocated for consideration of the **relevance and rationale** of/for a selection of additional AI approaches that you present i.e., these marks are awarded to

the extent that your report's arguments for and against the adoption of selected AI techniques for dealing with real-world versions of the types of problems considered in Parts 1, 2, and 3 are clear, compelling and creative.

- **5 marks** will be allocated for consideration of the **design and detail** of these approaches i.e., these marks are awarded to the extent that the description and discussion of AI techniques is technically sound and demonstrates a good grasp of the way that they work
- **5 marks** will be allocated for good writing and presentation. i.e., these marks are awarded to the extent that the writing is clear, concise, and articulate, the report is well structured and free from padding or waffle, and any diagrams, pseudocode, etc., are informative and well presented.

Note: Given the word limit, the report is not expected to be exhaustive in any sense. Rather it is expected to pick an angle and do a good job of considering that angle in detail and in depth. A deliberately under-described and incomplete list of possible "angles" that could be taken:

- An adversarial version of the problem in which agents are competing with each other
- A version where agents have to both collaborate to learn from each other and compete with each other to maximise their individual rewards in scenario where the agent who solve the maze first receive the highest reward
- A repeated version of the problem
- A dynamic version of the problem where obstacles, charging stations and/or oracle can move about randomly (or deliberately flee) from an approaching agent.
- etc.

Note: The phrases "real-world versions of the types of problems considered in Parts 1, 2, and 3" or "real-world search problems" should be understood to mean resource-limited spatial search problems, e.g., search and rescue, environmental monitoring, etc. Feel free to consider one important real-world example in your report, or to consider a small number of related problems.

Note: The AI techniques that you select for discussion in the report could be techniques that were presented in any part of this unit, or techniques that were discussed in wider reading that you have undertaken. Show that you understand your selected techniques in terms of their general use and overall utility and also in terms of the detail of how they work. It may be best to focus on a relatively small number of techniques and do each one justice, rather than spread your report over many different techniques without being able to describe each one to any great extent.

## 8 Marking Scheme

---

As explained in the preceding sections, marks are equally weighted across the 4 parts of the coursework.

In each of the three coding parts, marks will be equally distributed across five categories (with each category contributing a maximum of 5 marks) based on the extent to which your code improves upon the given skeleton code according to following scale: 0 for no improvement, 1-2 for a slight improvement, 3-4 for a substantial improvement, and 5 for a near optimal solution.

In cases where there is a potential tradeoff between primary, secondary and tertiary objectives, your marks for any higher priority criteria will serve as an upper bound on your mark for any lower priority criteria (as well helping to adjust the expected performance on the lower priority criteria).

### Part 1

- up to **5 marks** for the primary aim of minimising the number of **moves made** when executing direct paths in the standard and random grids;
- up to **5 marks** for the secondary aim of minimising the amount of **time taken** when finding direct paths in the standard and random grids;
- up to **5 marks** for also being able to find and execute **indirect paths** that allow agents to reach distant targets by refuelling at one or more charge stations en-route;
- up to **5 marks** for correctly handling some tricky or unusual **edge cases** that will be set up on specially hand-crafted grids;
- up to **5 marks** for demonstrating **good coding practice**.

### Part 2

- up to **5 marks** for the **primary** aim of **maximising the number of oracles** visited;
- up to **5 marks** for the **secondary** aim of **minimising the number of moves** made;
- up to **5 marks** for the **tertiary** aim of **minimising the time** taken;
- up to **5 marks** for correctly handling **edge cases** on hand-crafted grids;
- up to **5 marks** for demonstrating **good coding practice**.

### Part 3

- up to **5 marks** for a solution which allows a single agent to successfully leave the maze making reasonable moves. This means that the agent will not repeatedly explore routes which it has determined lead to a dead end and it will successfully leave the maze once it gets to the end.

- up to **5 marks** for a solution that supports 3 agents and makes effective use of them. This means agents will explore different paths and will ideally avoid dead ends discovered by each other. 3 agents forming a conga line and then exploring as though they were one would not be worth any marks for this section.
- up to **5 marks** for everything up to this point working with a variable number of agents, that can range from 1 to 10.
- up to **5 marks** for a successful exit strategy (i.e. if upon discovery of the end point by one agent, all the other agents efficiently navigate to it using near shortest paths).
- up to **5 marks** for efficiency of solutions in terms of runtime.

The essay part will be marked as follows:

#### Part 4

- **5 marks** will be allocated for consideration of the **strengths and shortcomings** of the approaches used so far in Parts 1, 2, and 3. i.e., these marks are awarded based on your report's discussion of the extent to which the kind of algorithms developed during Parts 1, 2, and 3, may or may not generalise successfully to real-world search problems.
- **10 marks** will be allocated for consideration of the **relevance and rationale** of/for a selection of additional AI approaches that you present i.e., these marks are awarded to the extent that your report's arguments for and against the adoption of selected AI techniques for dealing with real-world versions of the types of problems considered in Parts 1, 2, and 3 are clear, compelling and creative.
- **5 marks** will be allocated for consideration of the **design and detail** of these approaches i.e., these marks are awarded to the extent that the description and discussion of AI techniques is technically sound and demonstrates a good grasp of the way that they work
- **5 marks** will be allocated for good writing and presentation. i.e., these marks are awarded to the extent that the writing is clear, concise, and articulate, the report is well structured and free from padding or waffle, and any diagrams, pseudocode, etc., are informative and well presented.

For more details and advice, please see the relevant sections of the documentation.