CS2910 Assessed Coursework 1

This assignment **must be submitted** by 23:59 on the 26th of February 2020.

Feedback will be provided by 18th of March 2020.

1 Learning outcomes assessed

This assignment covers some of the algorithms we covered on *uninformed search*. In particular, the outcomes assessed are:-

- knowledge and understanding of uninformed search strategies such as *breadth-first*, *depth-first* and *iterative-deepening*;
- application of uninformed search strategies to search both trees and graphs;
- implementation of uninformed search strategies in Prolog.

2 Instructions

You will submit your work electronically by means of the Department's submission script called `submitCoursework`. You may resubmit your script any number of times, though only the last submission will be kept. The submission will occur on `linux.cim.rhul.ac.uk` and the protocol is `submitCoursework <your directory>`. For example, assuming the directory with the solutions is `assignment1`, then you should submit by typing in the following command:

```
submitCoursework assignment1
```

and then follow the instructions accordingly. Your directory should contain the following Prolog files only (see detailed assignment specification later in Section 4):

- `breadth_first.pl`;
- `depth_first.pl`;
- `graph_search.pl`;
- `efficient_searches.pl`.

The files you submit cannot be overwritten by anyone else, and they cannot be read by any other student. You can, however, overwrite your submission as often as you like, by resubmitting, though only the last version submitted will be kept. Submission after the deadline will be accepted but it will automatically be recorded as being late and is subject to College Regulations on late submissions. Please note that all your submissions will be graded anonymously, so please do not put any references of your name or College identifier anywhere in your submission.

NOTE:

All the work you submit should be solely your own work. Coursework submissions are routinely checked for this. Any assessment offence will be investigated subject to the College regulations.

3 An *Uninformed Search* template in Prolog

This assignment is based on extending and/or modifying the contents of file `tree_search.pl`, which corresponds to the Prolog implementation of the basic tree search algorithm on slide 4 of *uninformed search* topic. Download this Prolog file from `Moodle` and store it in your working directory for this assignment. Its contents are shown for convenience below:

```
search(Paths, Solution):-
        choose([Node|Nodes], Paths, _),
        goal(Node),
        reverse([Node|Nodes], Solution).
search(Paths, Solution):-
        choose(Path, Paths, RestOfPaths),
        findall([NewNode|Path], expands(Path, NewNode), Expansions),
        combine(Expansions, RestOfPaths, NewPaths),
        search(NewPaths, Solution).

expands([Node|_], NewNode):-
        arc(Node, NewNode).
```

In the above program `goal/1` and `arc/2` are predicates used to define any given search problem. An assertion of the form `arc(d, e)` means that we can go from node `d` to node `e` but not the other way around. More comments on the arguments of `search/2` and their use are available on the `tree_search.pl` you downloaded from `Moodle`.

The following directories in `Moodle` are useful resources to this assignment.

- `Traces` - Contains two files showing how `tree_search.pl` behaves while searching for a solution using a breadth-first and depth-first strategy respectively. This will allow you to imitate this behaviour while implementing these strategies.
- `Tests` – Provides four files that you can use to test the search strategies you will be developing. You may find it useful to draw on paper the trees/graphs represented by the contents of these files, so that you can verify the solutions your programs find.
- `Outputs` – Gives examples of what output we expect your programs to produce, when you are running different strategies on the different test files. **It is important** to ensure your solutions generate exactly the same output as shown in the files contained in this directory, as this will help us automatically test your programs.

4 Tasks for this assignment

T1. From Tree Search to Graph Search

Use the `tree_search.pl` program you have downloaded as a template for searching. Different strategies can be defined either by specifying appropriately the predicates `choose/3` and `combine/3` (which are currently undefined), and by constraining the way nodes get expanded.

(a) Implement the *breadth-first* search strategy for trees by appropriately defining `choose/3` and `combine/3` in a file called `breadth_first.pl`. [**10%**]

(b) Implement *depth-first* search for trees by appropriately defining `choose/3` and `combine/3` in a new file now called `depth_first.pl`. [**10%**]

(c) Copy the `tree_search.pl` file to a new file called `graph_search.pl` and change the `search/2` definition to avoid loops by checking that new nodes that are expanded are not already in the path. [**10%**]

From the `Tests` directory on Moodle, use `test1.pl` and `test2.pl` to test (a) and (b) above, and use `test3.pl` and `test4.pl` to test (c) above.

T2. Searching Efficiently

Depth-first based on the graph search template is not *memory efficient* because it keeps all active branches while searching for a goal. A more efficient search for graphs looks at one path only and exploits the Prolog built-in backtracking mechanism. Create a file `efficient_searches.pl` and incrementally add to it the following.

(a) Implement `dfs/2` (depth-first search) – this strategy takes a *single path*[1] as input and then expands that path with a new node until the goal is found, in which case the solution is returned in the second argument. Query your program with:

```
?- dfs([a], X).
```

for each file in `Tests` and check you find the right solution. [**15%**]

(b) Depth-first search is not complete for infinite spaces, so the *depth-limited* search has been proposed to deal with this problem. Implement `dldfs/3` (depth-limited depth-first search) – this strategy takes a single path as input in its first argument and then expands it with a new node until the goal is found, provided the length of the path does not exceed a depth limit `D` (> 0) specified in the second argument. When a solution is found within the limit, it should be output in the predicate's third argument. Test your program with a range of queries of the form:

```
?- dldfs([a], D, X).
```

Use different values for `D` (e.g. `1, 2, 3, 4...`) to see how your strategy works with different depth limits for each file in `Tests`. [**15%**]

(c) Use the depth-limited search you defined in the previous task to implement `ids/4` (iterative-deepening search). This takes a path as input in the first argument and a range of depth limits from 1 to `Max` (> 0) in the second and third arguments respectively, and checks whether there is a solution for every depth in that range. If a solution exists, it outputs the solution to the fourth argument; otherwise, it increases the depth of the search and continues searching for the solution until the `Max` depth is reached. Test your program with a range of queries of the form:

```
?- ids([a], 1, Max, X).
```

[1] Unlike `search/2` that requires all the paths as a list of lists e.g. `[[c,b,a],[d,c,a],...[f,g,a]]`, `dfs/2` should operate on one list only e.g. `[c,b,a]`.

Use different values for `Max` (e.g. `1, 2, 3, 4...`) and try the query on each file in `Tests`. **[15%]**

(d) The problem with `ids/4`, as defined above, is that the range of depths is fixed and has to be supplied in advance of the search. Even if all solutions have been found at a depth `D` $\in$ {1,...,`Max`}, the strategy keeps searching and keeps on producing the same solutions until `Max` is reached. Implement `idsh/4` (iterative-deepening search with history) – a new strategy that takes a path as input in the first argument, a depth limit D (> 0) in the second argument and then finds all the solutions at that depth and adds them to a history, a list of lists stored in the third argument. The strategy deepens the search only if the solutions found at `D` are different from those of `D-1`; otherwise if they are the same and all the goals have been found, then it stops and returns the paths found one by one. Check your implementation by testing the following query:

```
?- idsh([a], 1, [], X).
```

As before, test your program with each file in `Tests`. **[25%]**

Marking criteria

- Full marks will be given for implementations that address the requirements of all the tasks and their sub-tasks as specified in this document.
- Marks will be allocated to the logic of the strategies proposed as well as their implementation using appropriate recursive definitions.
- Marks will also be allocated in solutions which show understanding of Prolog unification, especially on the use of lists, the use of existing primitives that manipulate and generate lists (like `append/3` and `findall/3`), including the relevance of all these to the specific search strategies.
- Code quality: indentation, comments, variable naming, use of '_' variables, and appropriate use of Prolog control operators (e.g. the cut operator (!)).
- It is expected that the files you will submit are created using Linux and not Windows, as the marking will be done on a Linux machine. The code should run in `SWI Prolog` version installed on `linux.cim.rhul.ac.uk`. Implementations in any other Prolog or programming language will not be accepted.
- Your code should compile successfully for full marks. If part of your code does not compile, then wrap it in a comment of the form:

```
/* Partial Code:
....
End of Partial Code */
```

  and we will try to mark any logic that is relevant to the required task.

KS-07/02/20.