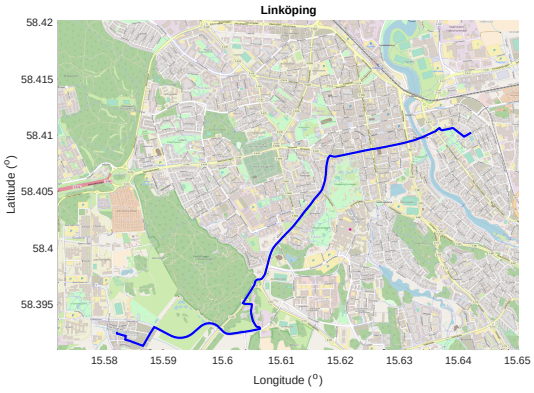
TSFS12 HAND-IN EXERCISE 1
Discrete Planning in a Structured Road Network

August 17, 2021

1 OBJECTIVE

The objective of this hand-in exercise is to gain fundamental understanding of the basic search algorithms for finding plans in finite search spaces using graph search algorithms. These graph search algorithms are fundamental tools in more advanced planning problems where, e.g., there are motion models of autonomous vehicles or planning under uncertainty. In this case, there is no uncertainty and the objective is to find routes in a map of central Linköping similar to a GPS planning tool or the map application on a mobile phone as illustrated in the figure below.

There are two versions of this exercise and for the first one, the objective is to implement different planners, evaluate their properties, and understand when and how they are appropriate. Some students have experience in implementing such basic search strategies before, and for those there is an alternative exercise where you investigate a real-time search strategy instead. You are free to choose which version you want to complete, *no matter your previous experience*.



It is allowed to discuss the exercises on a general level with other course participants. However, code sharing outside groups is not allowed. Moreover, both students in the group should be fully involved in performing all exercises, and thereby be prepared to answer questions on all subtasks.

See Appendix B for the extra assignment needed for higher grades. The extra assignment is done individually and is submitted by a document answering the questions, including suitable figures, plots, and tables. The document is to be submitted in PDF format. The code should also be submitted as a zip-archive. There is only pass/fail on the extra exercise and the document can not be revised or extended after first submission. It is not required to get everything correct to pass the assignment, we assess the whole submission. You are of course welcome to ask us if you have questions or want us to clarify the exercise.

Since these exercises examines this course, we would like to ask you not to distribute or make your solutions publicly available, e.g., by putting them on github. (A private repository on gitlab@liu is of course fine).

Thanks for your assistance!

2 PREPARATIONS BEFORE THE EXERCISE

Before doing this exercise, make sure you have read and understood the material covered in

- Chapter 2 in the book [2].
- Course lectures covering discrete search methods

To get the background code for the exercise, get the latest files from `https://gitlab.liu.se/vehsys/tsfs12` and familiarize yourself with the provided code. Start by running the code in the lab template file `main{.m, .ipynb, .py}`.

You are free to choose in which language to implement your planners, there are skeleton code available in Matlab and Python. If you choose Python, see Appendix A.3 for installation details or how to run the code in the student labs. The Python skeleton files will be available in both Notebook format (`.ipynb`) (`https://jupyterlab.readthedocs.io/`) or as python script files (`.py`) and you can choose to work with the format you prefer.

3 BASIC VERSION

To pass the basic version of this exercise you should implement, in a forward search mode, the following planners:

- Breadth First search
- Best First
- Dijkstra
- A*

The exercise is examined by submitting the following on the Lisam course page.

1. Runnable code. If your code consist of several files, submit a zip-archive.
2. A short document, that does *not* have to be formatted as a self contained report:
   - answers to the questions, including relevant plots, in Section 3.2, and other questions that you have encountered during solving this exercise.
   - a concluding discussion.

   Submit the document in PDF format. If you are using notebooks in Python, it is possible to submit your report in the form of a runnable notebook.

3.1 Planner Implementation Requirements

This section describes the specific requirements on your planner implementations. For this purpose, a `depth_first` planner is already implemented and all planners are required to have the same API defined as

```
planner(number_of_nodes, mission, f_next, heuristic, number_of_controls)
```

where the input arguments are

- `number_of_nodes`
  The number of nodes in the planning graph. Useful for pre-allocating space in planners.
- `mission`
  A struct (Matlab)/dictionary (Python) with the keys `start` and `goal` to represent positions for start and goal of the plan. Each point is represented also as a struct/dictionary with the keys `id`, `pos` (optional) where `id` is a node identifier and pos is longitude and latitude.
- `f_next`
  State transition function. For a given node x, the call

  Matlab

  ```
  [xi, u, d] = f_next(x);
  ```

  Python

  ```
  xi, u, d = f_next(x)
  ```

  returns possible next states xi (neighbours), the corresponding control that takes them there, and their corresponding distances d. This function is already implemented, see the skeleton file for details. Note, in this hand-in you do not need to consider the controls and here the controls are set to NaN. In later hand-in exercises, these will be extended to trajectories.
- `heuristic` (only used in `best_first` and `astar`)
  Planners `best_first` and `astar` needs a heuristic function. To define heuristics, you will find `osm_map.nodeposition` useful where again `osm_map` is an OpenStreetMap object and the function `latlong_distance`. See more in the code skeleton how to implement this.
- `number_of_controls` (not used in this exercise)
  Number of control signals. If there are no control signals, like in this exercise, set to 0. In the depth first implementation, this argument defaults to 0.

The planner should return a struct (Matlab)/dictionary (Python) with the properties

- `plan`: List/array of nodes in the plan
- `length`: Length of the plan
- `num_expanded_nodes`: Number of nodes expanded during planning
- `expanded_nodes`: List of nodes expanded during search
- `name`: Name of planner
- `time`: Time it took to find plan
- `control`: The controls for each step in the plan (not used in this exercise)

When implementing the planners, start with the code for the `depth_first` planner, only small changes are required for the other planners. If you are starting to make major changes, take a step back and think. For further details on available support code, see Appendix A.

3.2 Discussion topics, questions

This section summarizes a number of discussion topics and questions you should implement, investigate, and reflect upon to pass this exercise. Feel free to include other investigations and experiments you find interesting in the report.

Exercise 3.1. A discrete planning problem is defined by

1. A state-space $\mathcal{X}$
2. For each state $x \in \mathcal{X}$, a finite action space $U(x)$
3. A state transition function $f(x, u)$ that produces a state $x' = f(x, u) \in \mathcal{X}$ for every $x \in \mathcal{X}$ and $u \in U(x)$
4. Initial state x_I and goal state x_G

Determine for this route planning problem what $\mathcal{X}$ is and describe how $f(x, u)$ and $U(x)$ above relates to the provided Matlab function `f_next`.

Exercise 3.2. Implement the four required planners; `breadth_first`, `dijkstra`, `best_first`, and `astar`. Experiment with different types of missions, easy and more complex missions, and plot the resulting plans.

- Reflect upon the results with respect to length of plans and time to compute the plan. To illustrate, for 1 suitable mission,
  - plot plan lengths vs. planning times for the different planners
  - plot planning time vs. number of expanded nodes during search
- Explore a few different missions of different length and discuss the difference between A* and Dijkstra planners. Plot plan time against mission length. You can define your own missions or use the pre-defined missions given in `pre_mission`.

Exercise 3.3. After experimenting with different missions, reflect on your results and discuss

- properties of the different planners; what are the pros and cons?
- when are the different planners a good option, when are they not?

Exercise 3.4. The performance of A* relies heavily on the performance of the heuristic $h(x)$ and there are two main conditions; admissibility and consistency:

$$h(x) \leq h^*(x) \qquad \text{(admissible)}$$
$$h(x) \leq d(x, y) + h(y) \qquad \text{(consistent)}$$

where $h^*(x)$ is the optimal (and unknown) cost-to-go function, and $d(x, y)$ is the distance between nodes x and y. Describe consequences if each of these two assumptions are not fulfilled:

- what would happen if the admissibility assumption is not fulfilled?
- what happens if the consistency assumption is not fulfilled?

Exercise 3.5. Consider a labyrinth like mission/environment; explain why Euclidian distance is not a good heuristic for A* and Dijkstra search. Illustrate using a small example/figure.

4 ALTERNATIVE EXERCISE REQUIREMENTS

One problem with graph search algorithms, like A* explored in this hand-in, is that there is no way of predicting just how long a particular planning problem will take to solve. Therefore it can be difficult to use in a real-time application. For this purpose there are extensions, e.g., *just-in-time* planners that quickly can obtain a sub-optimal solution and then iteratively improve this solution until the user is satisfied with the solution quality or available time is up.

In this alternative exercise, you are expected to implement the algorithm described in the paper [3] (link in course git-repository)

> Likhachev, M., Geoffrey J. G., and Thrun S.. "*ARA*: Anytime A* with provable bounds on sub-optimality*", Advances in neural information processing systems. 2004.

and explore its properties. Before starting this exercise, read the paper and read the instructions in Section 3.

To pass the basic version of this exercise you should implement, in a forward search mode

- A*
- ARA*

The exercise is examined by submitting the following on the Lisam course page.

1. Runnable code. If your code consist of several files, submit a zip-archive.
2. A short document, that does *not* have to be formatted as a self contained report:
   - answers to the questions, including relevant plots, in Section 4.1, and other questions that you have encountered during solving this exercise.
   - a concluding discussion.

   Submit the document in PDF format. If you are using notebooks in Python, it is possible to submit your report in the form of a runnable notebook.

4.1 Discussion topics, questions

This section summarizes a number of discussion topics and questions you should implement, investigate, and reflect upon to pass this exercise. Feel free to include other investigations and experiments you find interesting in the report.

Exercise 4.1. Start by implementing an A* algorithm as in the basic exercise.

Exercise 4.2. A search with A* and an admissible and consistent heuristic $h(x)$ ensures optimal plans. Explore what happens in A* if you inflate the heuristic, i.e., multiply $h(x)$ with a factor $c > 1$ as

$$h'(x) = c\,h(x).$$

and instead use $h'(x)$ as heuristic. Run experiments on the map routing problem with different values of c and plot plan lengths and planning times. Comment on your findings.

Exercise 4.3. Which plans do the A* search converge to when the inflation factor $c \to 0$ and $c \to \infty$ respectively?

What can be said about the heuristic, or the planning problem, if the A* solution when $c \to \infty$ is close to the solution when $c = 1$.

Exercise 4.4. Describe the basic principles with the ARA* algorithm and how it can be used in a real-time application.

Exercise 4.5. Implement the ARA* algorithm from the paper and explore properties on sample missions.

Exercise 4.6. Quantify, in time, what is the gain when reusing previous computations when updating the inflation factor. Compare with A* and no computational reuse.

A SUPPLEMENTARY CODE AND SOFTWARE INSTALLATION

A.1 Queues

To implement the planning algorithms, three different queues have been implemented in both Matlab and Python

- LIFO
- FIFO
- Priority queue

A LIFO, Last In First Out, queue is a queue where the last element inserted into the queue is also the first to leave the queue. Similarly, a FIFO, First In First Out, is a queue where the first element inserted into the queue is the first to leave the queue. A priority queue has the property that there is a priority associated with each element and when extracting an element from the queue, the element with the lowest priority is taken from the queue.

Each queue has the following methods implemented

- insert
- pop
- peek
- size
- isempty

and to get help for the methods, write for example

Matlab

```
help PriorityQueue.peek
```

Python

```
help(PriorityQueue.peek)
```

Below is a small example how to create a `FIFO` queue and insert/extract values.

Matlab

```
>> q = FIFO();
>> q.insert(10);
>> q.insert(20);
>> q.peek()
ans =
     10
>> x = q.pop()
x =
     10
>> x = q.pop()
x =
     20
```

Python

```
In [1]: q = FIFO()
In [2]: q.insert(10)
In [3]: q.insert(20)
In [4]: q.peek()
Out[4]: 10
In [5]: x = q.pop()
In [18]: x
Out[18]: 10
In [19]: x = q.pop()
In [20]: x
Out[20]: 20
```

Below is a small example in Matlab how to create a `PriorityQueue` and insert/extract values. First, create the queue and insert values into the queue with some priorities

Matlab

```
% Create empty queue
>> q = PriorityQueue();
% Insert value 1 with priority 10
>> q.insert(10, 1);
>> q.insert(20, 2);
>> q.insert(5, 3);
>> q.insert(7, 4);
```

Python

```
# Create empty queue
In [1]: q = PriorityQueue()
# Insert value 1 with priority 10
In [2]: q.insert(10, 1)
In [3]: q.insert(20, 2)
In [4]: q.insert(5, 3)
In [5]: q.insert(7, 4)
```

To check the size, write

Matlab

```
>> q.size()
ans =
     4
```

Python

```
In [6]: q.size()
Out[6]: 4
```

To peek and see which is the next value to leave the queue, and then get the element form the queue

Matlab

```
>> [prio, value] = q.peek()
prio =
     5
value =
     3
>> [prio, value] = q.pop()
prio =
     5
value =
     3
>> q.size()
ans =
     3
```

Python

```
In [7]: q.peek()
Out[7]: (5, 3)
In [8]: prio, value = q.pop()
In [9]: prio
Out[9]: 5
In [10]: value
Out[10]: 3
In [11]: q.size()
Out[11]: 3
```

A.2 OpenStreetMap

In the exercise code there is a class `OpenStreetMap` that represents maps. You don't have to use extensive functionality, mainly for defining missions, plotting plans, and for defining the search heuristics. Code for loading maps, defining missions, and plotting plans are included in the provided skeleton files. Some main properties and methods are described briefly below, for more details see class documentation.

The main information used for planning is a distance matrix. If the map has n nodes, the distance matrix $\mathcal{D}$ is an $n \times n$ matrix with path distances between *neighbors* as entries. Entry at row i and column j corresponds to the distance between node i and j. If the nodes are not neighbors, the entry is zero. The distance matrix can be accessed using the `distanceMatrix` property of the map object. In code, the (i, j) distance is accessed as

Matlab

```
osm_map.distancematrix(i, j)
```

Python

```
osm_map.distancematrix[i, j]
```

The distance matrix, due to its size, is represented as a sparse matrix.

In A* and Best-First search, there is a need to compute distances between nodes. For that purpose, there is a dictionary property `nodeposition` with latitude and longitude of a node. In Python and Matlab, the position is obtained by

Matlab

```
osm_map.nodeposition(idx)
```

Python

```
osm_map.nodeposition[idx]
```

where `map` is the `OpenStreetMap` object and `idx` is the node index. There is also a function `latlong_distance` that computes the distance in meters between to points given by latitude and longitude.

The function `load_osm_map` can be used to load the map. If there is a binary mat (Matlab)/pickle(Python) it will read that. If that file is not available, the OpenStreetMap XML-file is parsed and this will takes some time (could be a couple of minutes). It is recommended to read the binary files distributed with the exercise to ensure consistent numbering of nodes over different versions of Matlab/Python.

The main class methods, there are others that you can explore, that are useful are:

- `info()`
  Print basic information about the map.
- `getmapposition()`
  Take input from a map object by clicking in the map, and return the closest node index. This is useful for specifying start and goal nodes for the planning task.
- `plotmap()`
  Plot the map.
- `plotplan(plan)`
  Plot a plan, given as a list of node indices. If you have an array of nodes visited during search, `node_list`, and want to visualize those nodes, you can use `plotplan(node_list, '.')`.

To illustrate the use of some of the functions, the following commands loads the map defined in the exercise and displays some basic information.

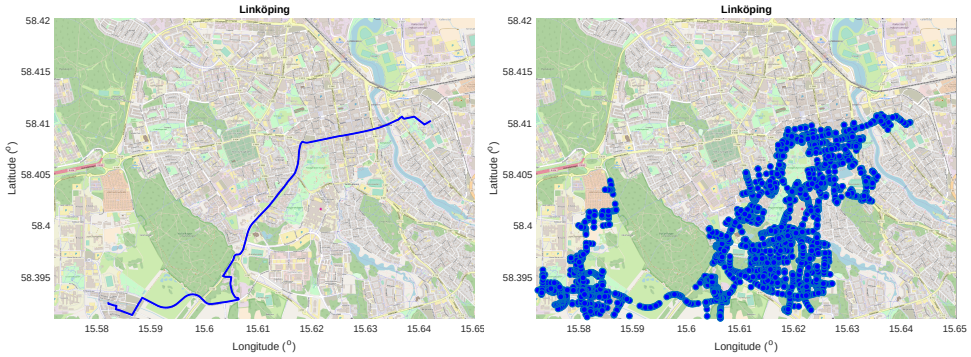
```
%% Read map information
>> mapfile = 'linkoping.osm';
>> figurefile = 'linkoping.png';
>> osm_map = load_osm_map(mapfile, figurefile);
>> osm_map.info()
OSMFile: ../Maps/linkoping.osm
Number of nodes: 12112
Number of roads: 2977
Map bounds
  Lon: 15.572100 - 15.650100
  Lat: 58.391000 - 58.420200
  Size: 3246.9 x 4545.8 (m)
Figure file: ../Maps/linkoping.png
  Size: 2296 x 1637 (px)
```

To use a planner, in this case the `astar`, and then plot the map with the corresponding plan, the following commands can be used which produces Figure 1-a.

```
>> astar_plan = astar(num_nodes, mission, f_next, cost_to_go);
>> osm_map.plotmap()
>> hold on
>> osm_map.plotplan(astar_plan.plan, 'b', 'linewidth', 2);
```

```
>> hold off
>> xlabel('Longitude (^o)')
>> ylabel('Latitude (^o)')
```

The corresponding commands in python are similar. The `plotplan` class method can



(a) (b)

Figure 1: Figure (a) shows a sample plan and figure (b) visited nodes during the search.

also be used to illustrate nodes visited during the search. Figure 1-b is produced using the command

```
>> osm_map.plotplan(bf_plan.visited_nodes, 'o', 'markerfacecolor', 'b')
```

A.3 Python installation

To run the python code you need Python3, (optionally) jupyter, and some additional packages installed. If you don't have Python installed, you can get it from `http://www.python.org`. Packages are typically installed using either a package manager like Conda (`https://www.continuum.io/downloads`) or using virtual environments (`https://docs.python.org/3/tutorial/venv.html`). Here, virtual environments will be described.

A.3.1 In the student labs

In the student labs, to activate a python interpreter with a correct version and all needed packages installed, start a terminal and activate the virtual environment by (% is the prompt and should not be typed)

```
% source /courses/tsfs12/env/bin/activate
```

Note that you have to do this every time you want to work in the virtual environment and in each terminal (or add the command to your startup scripts).

A.3.2 Private computer

If you want to work on your private computer, full installation instructions for virtual environments are shown below.

Open a terminal and create a virtual environment called `env` (you can name it whatever you want)

```
% python3 -m venv env
```

This will create a directory `env` and you only have to do this once. Then, activate the virtual environment by

```
% source env/bin/activate
```

For Windows you instead run

```
% env\Scripts\activate
```

Again, this you have to do every time you want to work in the virtual environment (or add the command to your startup scripts). The first time, you also want to install the required packages using the `pip` command by

```
(env) % pip install ipython numpy jupyter jupyterlab matplotlib scipy
```

If you choose to work with python scripts, open `main.py` in your favorite editor or IDE and start solving the exercises. Good choices for IDE include Visual Studio Code (`https://code.visualstudio.com`) and PyCharm (`https://www.jetbrains.com/pycharm/`).

If you choose to work in Jupyter Notebooks, start JupyterLab by

```
(env) % jupyter lab
```

and the development environment should appear in your web browser. Locate the file `main.ipynb` in the `code/python` directory and start exploring.

B EXTRA ASSIGNMENT

It is typically not feasible to pre-compute plans for any possible mission, since that would mean pre-computing n^2 plans where n is the number of nodes in the graph. However, consider the case where the goal of the mission is predetermined, say a central location in a distribution network or a train terminal, and we want to efficiently plan paths to that determined goal.

This extra assignment explores how methods from the basic assignment can be used for this purpose and also connect the results to more general optimization theory, and dynamic programming in particular. Therefore, before starting to solve the exercise, read pages 1–12 in the book [1][1]

> Bertsekas, Dimitri "*Reinforcement learning and optimal control*". Belmont, MA: Athena Scientific, 2019.

You can find skeleton material, including pre-defined missions you can start experimenting with, in the files `extra.{m, ipynb, py}` for Matlab and Python respectively.

Exercise B.1. Show, in detail, how the path planning problem in this hand-in corresponds to the deterministic dynamic programming problem introduced in Section 1.1.1 in [1].

Exercise B.2. Modify the basic Dijkstra algorithm to compute the cost-to-go function for a given goal node x_g. Thus, make a function that determines the optimal cost to go from node x to goal node x_g,

$$\text{cost-to-go}(x), \quad x \in \mathcal{X},$$

where $\mathcal{X}$ is the set of all nodes. As a goal node, you can for example use the goal node in the pre-defined missions.

Time how long it takes to pre-compute this function.

Exercise B.3. Write a function that computes a plan from a given start node using the pre-computed cost-to-go function and the state-transition function (`f_next` in Section 3.1. Also include timing functionality.

Exercise B.4. Ensure that the plans computed by Dijkstra, A*, and the new plans based on pre-computed cost-to-go function are equivalent, both in terms of node visits and length of plan.

Illustrate/plot how the planning times compare for the three planning approaches. You can use the pre-defined missions, but feel free to experiment with other missions.

Exercise B.5. Describe how your function in Exercise B.3 relates to the construction of the optimal sequence described on p. 12 in [1].

Exercise B.6. Comment on your solution and how it relates to the dynamic programming algorithm for deterministic finite horizon problems on p. 11 in [1].

1 An electronic version of the text can be found at https://web.mit.edu/dimitrib/www/RL_1-SHORT-INTERNET-POSTED.pdf and further information about the book is available at the authors page https://web.mit.edu/dimitrib/www/RLbook.html.

REFERENCES

[1] Dimitri P Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific Belmont, MA, 2019.

[2] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[3] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in neural information processing systems*, pages 767–774, 2004.