

Complex Planning and Optimization Problems

Part 1: Is it, though?

The trouble with advanced modules ...

Every project is different.

One-Size-Fits-None Learning Units.

- Specific Problems
- Specific Algorithms

The solution: Focus on best practices

- This means it's up to you to research specific algorithms.

In an environment like CODE, it is hard to organize good learning units for advanced modules. The issue is that students have complete freedom in what they want to do in their projects, meaning two projects may differ in what problem they solve, which criteria they have for an optimal solution, which algorithms they test, ...

This means a learning unit that handles specific problems or algorithms has the danger of being irrelevant.

My solution for this is to not focus on specific problems or algorithms but rather to focus on general methods such as clean code or experimentation to help you apply them to your learning unit.

Spot the difference...

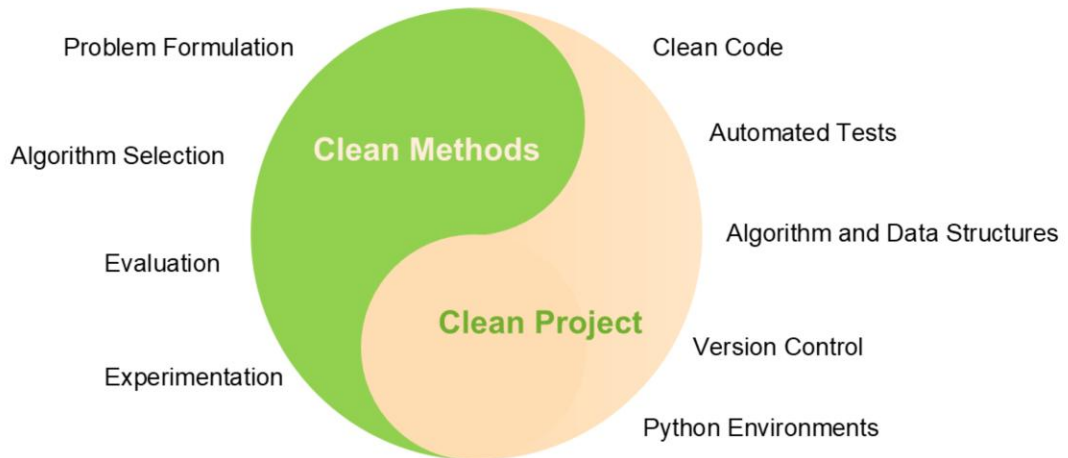
- Appropriate formulation of a **planning problem** in terms of:
 - State Representation
 - Actions
 - State Transition Model
 - Planning Goal
- Quality criteria (desirable properties of the search algorithm)
- Selection of a search algorithm for solving this **planning problem**, including appropriate methods for reducing and prioritizing the state space.
- Analysis of the performance of the selected algorithm with respect to the **planning goal** and quality criteria.

- Appropriate formulation of an **optimization problem** in terms of:
 - Input variables and types
 - Optimization goal as function over the input variables.
- Quality criteria (desirable properties of the search algorithm).
- Selection of a search algorithm for solving this **optimization problem**, including appropriate methods for reducing and prioritizing the state space.
- Analysis of the performance of the selected algorithm with respect to the **optimization goal** and quality criteria.

The images in this slide are taken from the module description of the Planning and Optimization module. As you might notice, they are quite similar. They both are set around a complex problem where students have to make good design decisions regarding how to formulate the problem, define selection criteria, select appropriate algorithms and analyse their performance.

This learning unit handles best practices around these steps and thus is applicable to both modules.

Goal of this learning unit: Best Practices



The goal of this learning unit is to provide you with best practices that you can apply to planning and optimization.

We will look at this from two complementing sides:

- **Clean Methods:** this means applying clean methodology to your experiments. It involves all activities of the two topics, including formulating a formal problem, defining success criteria, selecting algorithms and evaluating them. A lot of this is going to involve defining executing and documenting experiments.
- **Clean Projects:** this means having a project that is easy to understand and maintain. It focuses on best practices in software engineering, such as clean code, automates testing, version control, etc. and how they apply to experimentation projects.

Organizational notes.

We will use a running example.

- Project can be found [here](#).
- Feel free to copy / use parts if they are helpful to you.

No Homework

We will not talk about...

- Specific algorithms
- Specific libraries

There are a few other organizational notes you should be aware of.

The project will be based on a running example. Based on a high-level goal, we will search for datasets, evaluate different models, select hyper parameters and evaluate our resulting model.

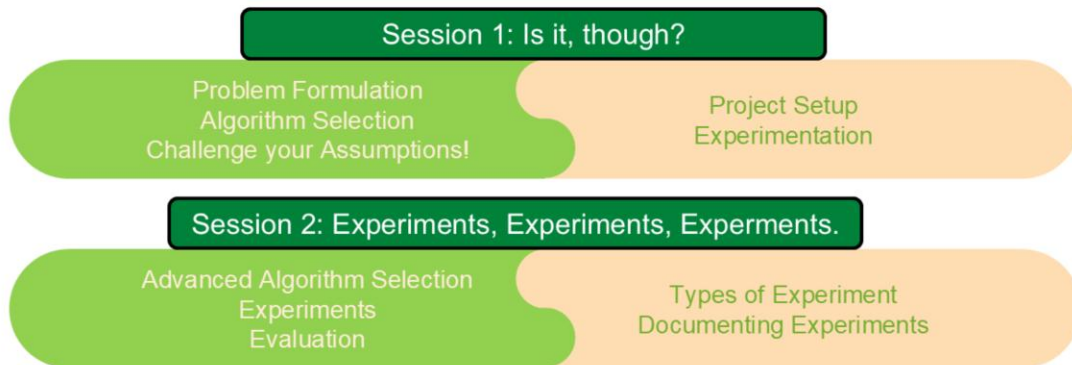
This is implemented in the linked git repository. Besides serving as a running example, this repository has been created as a resource for you to refer to. We will introduce the example later.

While looking through the project, you should be aware that we designed this project with the goal to be a good learning resource. This of course has influenced the design of the project. For example, some parts are a bit over-explained and contain more comments than strictly necessary.

The learning unit will contain some mandatory homework. The homework will be light touch (two to four hours maximum) and will be required so we all are on the same page in our discussions.

This learning unit has been designed to focus on best practices surrounding machine learning. We will not focus on specific models, the mathematics behind them or libraries you can use. There are good learning resources out there for these and you should pick whichever fit the problem you intend to solve.

Sessions



The learning unit is split into three Sessions.

In Session 1 we will lay the groundwork. We will start out with a goal, search for datasets, and explore and process the found dataset. At the end of this session we will have data and a good idea of what machine learning problem we want to solve. On the clean project side, we will focus on how a machine learning project can be set up and how machine learning files interact with standard Python processes.

Session 2 will focus on experimentation. We will select a model and find appropriate hyper-parameters for it. We will do this in three separate experiments, which illustrate three types of experiments you may come across while doing machine learning. On the clean project side, we will talk about how to plan and document experiments.

Session 3 will focus on evaluation of results and utilization of the resulting model. We will talk about how we can evaluate and interpret our model and how we can store and use the result of our learning in other software applications.

Sessions

Session 1: Is it, though?

Part 1: Problem Formulation

Part 2: Algorithm Selection

Part 3: Challenge your Assumptions

Today we will discuss the four parts mentioned in the slide.



The diagram is a rectangular frame divided into two main regions by a large, light green shape on the left. The right region is light orange and contains the text 'Project Setup'. The left region is light green and contains the text 'Project Goal', 'Success Criteria', and 'Quality Criteria'. A dark green rounded rectangle with a black border is centered horizontally, overlapping both regions, and contains the text 'Part 1: Problem Formulation'.

Project Setup

Part 1: Problem Formulation

Project Goal

Success Criteria

Quality Criteria

We are Detectives!



The goal: Try to find out if Person A has been in communication with Person B

The Complication: They may have used a third person to obfuscate their tracks.

The Evidence: We traced emails and know who has been writing to whom.

Side-Goal: Time is of the essence!

We will use a toy example for this learning unit as it gives us something practical to discuss and showcase.

For this purpose we will assume that we are detectives who try to find out if person A has been in contact with Person B directly or indirectly (by talking to intermediaries).

Let's assume we want to solve this type of question repeatedly with minimal overall time.

We are Detectives!



The Dataset:

- <https://snap.stanford.edu/data/email-EuAll.html>

Let's look into it → `data_exploration.ipynb`

As basis for this problem we will use the email EuAll dataset. This dataset has been collected and represents contacts between researchers.

We look into the dataset in more detail in notebook “`data_exploration.ipynb`”

Problem Formulation

Input:

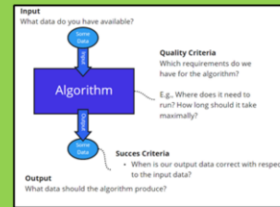
- G: Directed Graph, no node information
- A: Node
- B: Node

Output: True/False

Success Criteria:

- True => there is at least one path from A to B in G
- False => there is no path from A to B in G

Quality Criteria: minimize time to calculate output on average over all runs.



Let's formulate this as a concrete problem we can try to solve. This problem is defined in terms of an AI algorithm problem as we also use in the AI basics module.

The input to the algorithm we are searching is the graph we are searching in, along with nodes A and B.

The output of the algorithm should be True or False. The output is considered correct if it is True if and only if there is a path from A to B.

On the side of quality criteria we would like to find the algorithm that finds the path in the minimum amount of time on average.

Setting up the project

How do you set up a normal software project?

- Version control, issue tracking, project management
- Folder structure, clean code conventions, tests
- Tech choices, readme, venv

All of these apply to Experimentation projects as well.

- The project we use in this learning unit is an example.

Once you start your project, you should also set up a source code repository. There are quite a lot of best practices, tools and processes related to a software project. The slide names a few. While experimentation project might feel a little different than building a software system, there is no reason not to apply these best practices.

This may be a good point to look over the repository of the example project. It showcases one possible setup for an experimentation project. You don't know all of the content yet, but you can do what you should always do with a project on first glance: look over the Readme file. Try to understand the project scope and it's folder structure.

Just as with normal Python projects, there are different ideas on how to set up a project in a clean way (e.g., whether code should be in a dedicated source folder). Generally, you should find a good practice for you and your team and stick to it. We will go through more parts of the project as they become relevant to what we are talking about.

Aren't Notebooks a Machine Learning thing?

What is a notebook?

- executable code snippets
- markdown
- rich in-place output

When is a notebook appropriate?

1. You want to play around and experiment
2. Communication to an audience
 - Your customer, your project team, fellow researchers, your teacher ...

You might know notebooks from the Data Science / Machine Learning community. They are used there so often because these communities are often concerned with conducting and documenting particular experiments, rather than building a software system. This is also the reason why you may find them useful in planning / optimization projects.

In short, a notebook is an environment with cells containing code that can be executed. This is not dissimilar to the REPL mode of Python, where you enter commands and see the results. However, it is more sophisticated. It can execute whole blocks of code rather than single lines. And it can show complex output, like graphs, instead of just strings. This makes it very useful for data science, where you can get your graphs in the same document as your source code.

Notebooks can also be annotated extensively with Markdown text to write sophisticated texts with tables and figures, if you need to.

When do you use notebooks? There are a few occasions where they shine:

- You can use a notebook as an experimentation environment for yourself. For example, it may make sense to keep a notebook file out of version control to just play around with things before you set them into stone and put them into your more permanent code. This can be a good way to figure things out. However, the resulting notebooks tend to be messy and hard to read / understand so it's often not a good idea to put them into the project proper.
- You can use notebooks as a communication tool. This is making extensive use of

their markup capabilities to essentially write an essay that is complemented with source code. Your audience may vary. Maybe you want to document a decision for your project team. Maybe you want to report a result to your customer. Or maybe you want to document an experiment for a research paper. Or maybe you want to teach method to someone.

This last point is why so many online tutorials rely on notebooks. It's an easy communication tool for teaching. It's also why the example project contains a few more notebooks than would maybe be necessary: they are better ways to communicate certain information to you than documented source code or slides.

How to set up notebooks cleanly?

Use an IDE that can edit both notebooks and normal python

- VSCode is highly recommended

Same Python environment for notebooks and normal Python!

Adjust Notebook Working directory

- Needs to be root module of your project to find Python files

Notebooks and Git are slightly awkward

Setting up notebooks cleanly can be a bit tricky. Generally, what I recommend is to set up an environment in which you can easily edit and run traditional Python files as well as notebooks. This way you get a coherent programming environment for your whole project.

One IDE that does this well is VSCode. It has plugins for Python files and Jupiter notebook and can be made to use the same Python environment for both. Essentially, you can create a virtual environment, install all of your libraries in it and then use it for both your normal python and your notebook executions and always be certain to use the correct versions of your libraries.

When you use notebooks, you may have to do some extra work to be able to import your Python files. The default working directory of a notebook is the folder it is placed in. If your notebook is in your project root folder, this should work well. If have decided to have a different structure – maybe a dedicated notebook folder, like our example project, then you will have to work a bit. You can find example code at the beginning of each of our projects that automatically sets the working directory to the correct folder.

Notebooks also are compatible with version control now (this used to be somewhat of an issue). You can do all of your usual git operations on notebooks. The only thing you need to be aware of is that the output is also in version control. This means if you rerun the notebook it will think there is an update that could be committed. It's easy to

clutter your git repository by committing the same output of running the same cells at different times. Be mindful of that and only commit notebooks if they have intentional changes.

Explanations in Notebooks

Candidates

Part 2: Algorithm Selection

Experiments

Data Structures

Algorithms

What can we derive from the Problem?

This is a Pathfinding Problem.

This is not a shortest Path Problem.

This is not an informed search problem.

Applicable Algorithms:

- Breadth First Search (BFS) ($O(N + E)$)
- Depth First Search (DFS) ($O(N + E)$)
- Dijkstra's Algorithm ($O(N + E) \log N$)

N = Nr Nodes
 E = Number edges

Now that we have a problem, we can analyze it to derive good algorithms. From the problem alone, we can gain a few insights:

- The problem of deciding whether a path exists between A and B is likely solved by searching for a path from A to B and returning True if such a path exists and False if not. Thus we should be searching for algorithms that can calculate paths.
- This is not a shortest path problem. Any path is sufficient to answer True or False.
- This is not an informed search problem. Our nodes only have numbers but no additional information. It is unlikely that the id of the node contains enough semantics to be useful in the search. This means we cannot apply informed search algorithms.

According to these insights, the obvious uninformed path finding algorithms are breadth first search and depth first search. We could also apply Dijkstra's algorithm (assuming a constant edge length of 1).

Algorithm Selection Experiment.

The candidates: BFS, DFS, Dijkstra's

The setup:

- Run each algorithm on (the same) 1000 random planning problems
- Record average execution times.

The experiment: `basic_algorithms_experiment.ipynb`

Now that we have identified a first set of suitable algorithms, we can run an experiment to see which one works best.

The experiment runs all algorithms on the same 1000 planning problems in order to decide which one is fastest on this dataset. Since we are interested in the fastest algorithm, we will compare the three based on the average execution time on these 1000 problems.

The experiment and its results can be seen in the linked notebook.

Writing an Experiment Notebook

Same as in an essay / thesis

1. State your goals
2. Define procedure
3. List results
4. Discuss results in context of goals

Remember: You write for an audience!

- Explain things they don't know
- Describe assumptions and conclusions explicitly
- Choose appropriate language

Now that we've looked into our first notebook, let's talk a bit about how to write a good notebook for an experiment.

Structure-wise, this should not be much different from any other experiment summary, meaning it should...

- Explicitly state the goals of the thesis. This could be a question or a hypothesis.
- Define the experiment procedure in enough level of detail for someone to repeat your experiment
- List the results. This should be done without interpretation.
- Discuss the results. The discussion should be directed towards the goals you defined in the beginning.

This refers to the second use case for a notebook, where you try to communicate something to an audience. Essentially, you should regard your notebook as an essay or text. You should state the purpose and goals of the notebook clearly in the beginning and circle back to them by the end. The source code in this type notebook is there to support your text – by producing results or providing mini-experiments the reader can carry out. Remember that you write for an audience. Make sure that you're writing in a way they can understand.

Algorithms and Data Structures matter

What choices do we need to make?

- Graph Representation
- Open and Closed List
- Algorithm Variations

What do we optimize for?

- Our quality criterium is time!

One design decision that the experiment has glanced over is how the tested algorithms have been implemented in the first place.

Implementing an algorithm is not always a cut and dry thing. Choosing different variations of the algorithm or different data structures during the implementation can impact the performance. In this project we made the deliberate choice not to use any frameworks but to implement these algorithms ourselves so we can illustrate these choices. Here are the things we should consider:

- How do we represent our graphs? There are two general options: Adjacency Matrix and Adjacency list. Both have different time complexities depending on how one uses the structure.
- How do we represent data within the algorithms? In the planning algorithms we tested the main design decision is which data structure to use for the lists keeping track of open and closed nodes.
- Which variation of the algorithm do we take? There may be different ways to implement the algorithm with different run times.

What we are optimizing for depends on our problem. In our running example, time is our quality criterium. In other situations, memory usage or other factors may be important.

Graph Representations

What choices do we have?

	Adjacency Matrix	Adjacency List
Memory Usage	$O(N^2)$	$O(N + E)$
Connection check	$O(1)$	$O(E_n)$
Outgoing edge iteration	$O(N)$	$O(E_n)$

N = Nr Nodes
 E = Number edges
 E_n = Number edges from node n

What is important for our use case?

Which one do we choose?

With respect to the graph representation, the basic choices we have are adjacency lists and adjacency Matrices.

The table above summarizes some of the properties that may be relevant for a selection. But which properties are important for our use case? Since we want to optimize for time, the memory usage is not relevant. And since we want to apply planning algorithms, which are based on the search tree, we will frequently have to iterate all outgoing edges from a node.

Accordingly, we should decide based on the outgoing edge iteration property.

Adjacency lists have the better performance here. As we have seen in our dataset, the number of outgoing edges is usually fairly small and always smaller than the number of nodes (because there are not multiple edges between two nodes).

For this reason, we will choose adjacency lists.

List Data Structures

Which data structures do we use in our algorithms?

- Open List – Keeps track of pushing and popping open nodes
- Closed List – keeps track of which nodes we have already visited.

Which operations are important?

Open	BFS	DFS	Dijkstra
Push	X	X	X
Pop	Oldest	Newest	Smallest

Closed	BFS	DFS	Dijkstra
Push	X	X	X
Check	X	X	X

Which Data Structures to choose?

Which data structure choices do we need to make? The most obvious ones in BFS, DFS and Dijkstra's algorithms are the two list data types:

- The open List is used to keep track of nodes that have been seen but not visited in detail.
- The closed list is used to keep track of nodes that have been visited. It is used in non-tree search spaces to avoid going down the same path twice.

Looking at how these two lists are used in our algorithms shows which operations we should optimize for. Summarizing, we can see the following similarities and differences:

- All algorithms use mainly push and pop operations in the open list. They differ in which element they want the pop operation to return.
- All algorithms want to use the closed list like a register to which they push visited nodes and check whether a node has been visited.

Based on these, we can narrow down the options:

- The closed list is probably best implemented by an array-like structure. We know the number of nodes and don't care about memory usage. This means we can profit from constant access time in the array.
- The open list is some kind of queue, but different types are appropriate for the three algorithms:
 - BFS: First-In-First-Out Queue
 - DFS: Last-In-First-Out Queue (Stack)

- Dijkstra: Priority Queue

-

List Data Structures

Which Data Structures to choose?

	Open List	Closed List
BFS	FIFO-Queue	Array
DFS	LIFO-Queue	Array
Dijkstra	Priority Queue	Array

But what if there are multiple implementations?

LifoQueue Lists Queue
deque PriorityQueue
heapq SimpleQueue

Experiment Time! → `data_structures_experiment.ipynb`

Based on these, we can narrow down the options:

- The closed list is probably best implemented by an array-like structure. We know the number of nodes and don't care about memory usage. This means we can profit from constant access time in the array.
- The open list is some kind of queue, but different types are appropriate for the three algorithms:
 - BFS: First-In-First-Out Queue
 - DFS: Last-In-First-Out Queue (Stack)
 - Dijkstra: Priority Queue

These options are still theoretical. Once you work with a specific programming language you may have different implementations to choose between. For example, Python has multiple data types that could serve as a Queue. How do you choose between? Generally, you do so based on the operations we already identified previously. And if you can't find any literature that compares these data types, you can always experiment yourself.

Notebook `data_structures` contains an experiment to compare the different Queue implementations against each other.

Algorithm Variations

In which ways can you implement a simple BFS algorithm?

Recursion or Iteration?

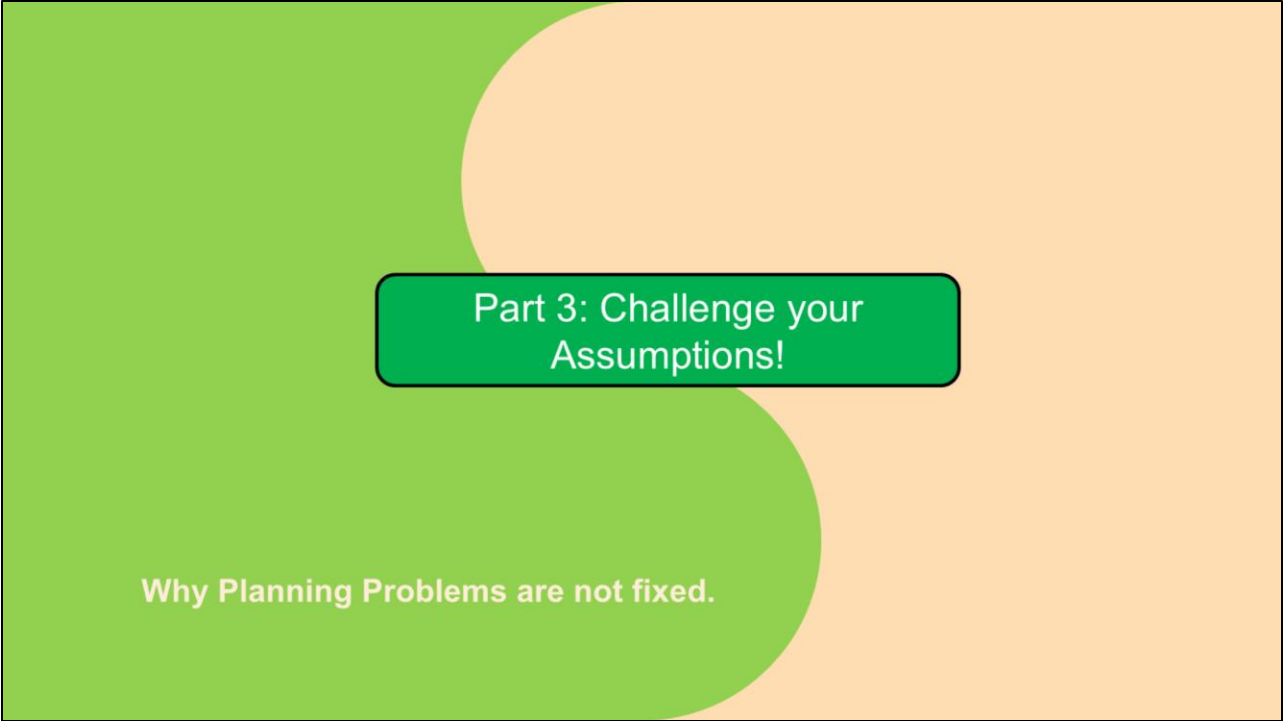
- Recursion is generally slower
- Exceptions exist (e.g., if avoiding recursion requires using expensive data structures)

When in doubt ... Experiment!

You will often have different options for implementing algorithms. Here are some examples of choices taken in our project:

- Do we implement search algorithms using loops or do we use recursion?
Generally, recursion is slower than iteration, but it may be faster in cases where the recursion allows you to avoid using a costly data structure.
- Specifically for Dijkstra, there is a special case in which we find a path that is shorter than an already known path, in which case we would have to substitute the node with the old one. We now have the option of removing the old path from the priority queue or not. Not removing it will make the queue longer and influence the time taken for pushing elements to the queue (since this time is $O(\log(|\text{queue}|))$). But removing it from the queue would require linear time or a more complex data structure that has logarithmic search complexity.

In our experiments we chose to avoid recursion and not remove elements from the open list. We have not verified this with experiments – mostly because these experiments would be redundant and not add much to the learning unit. If you are uncertain in your project it is absolutely viable to perform these experiments.



Part 3: Challenge your Assumptions!

Why Planning Problems are not fixed.

It seems like our problem is solved.

But is it really this easy?

	BFS	DFS	Dijkstra
Average Time	101 ms	76 ms	206 ms

Did we forget something?

Did we make some wrong assumptions?

Is this slide designed to make you paranoid?

Looking at our main experiment it seems like DFS is our clear winner. We could call it quits here and declare this algorithm the winner.

But how certain are we that we are finished? Is there maybe something we didn't test? Or something we could do to squeeze out even more speed?

This slide is designed to make you paranoid because you should always be. Usually your job only starts after testing the obvious candidates as that is the point where you start thinking about custom algorithms or problem-specific optimizations.

Let's challenge our assumptions...

This is a Pathfinding Problem. ← Is it, though?

This is not a shortest Path Problem. ← Could it be?

This is not an informed search problem. ←

Applicable Algorithms: ← Are there others?

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Dijkstra's Algorithm

Looking back at an earlier slide, we can actually see several assumptions that we could potentially challenge. We'll go through them in the next slides.

Are there other uninformed search algorithms?

What about ...

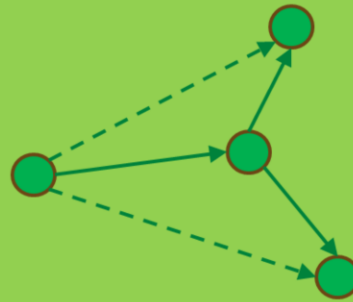
- Depth-Limited Search
 - DFS with maximal depth
 - Likely not helpful
- Iterative Deepening Depth-First Search
 - DFS with maximal depth.
 - If not goal, restart with higher maximal depth.
 - Possibly helpful
- Bi-Directional Search
 - Concurrently search from start and goal
 - Possibly helpful

There are several basic algorithms we have not implemented for this experiment. If this was a serious project we would have likely at least tried Bi-directional search. Depth limited search really is only useful if we can determine a maximum search depth. Considering DFS was more efficient than BFS in our experiment a too deep search depth is unlikely to be an issue. This also impacts the usefulness of iterative deepening, though it could be tried.

Is this really a Pathfinding Problem?

Path finding is certainly one option.

But there could be other ones.



Alternative: Transitive closure

- 'Complete' the graph by adding closure edges
- Needs to be done only once.
- Example: Floyd Warshall Algorithm ($O(N^3)$)

While path finding is certainly one option to solve this problem, it may not be the only alternative. An example is given on the slide: transitive closure algorithms. These algorithms work by refining the graph and adding edges that represent pathways of edges. These algorithms typically have a higher complexity than pure pathfinding algorithms since they need to check paths from all nodes to all other nodes. However, they only need to be run once. Afterwards, connections can be looked up in constant time (if the graph is represented as adjacency matrix).

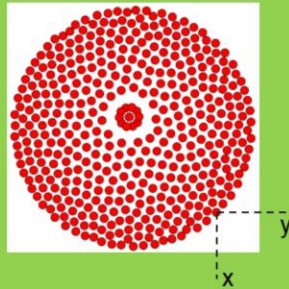
If we ask a large amount of questions, such an algorithm may be helpful.

An example for such an algorithm is the Floyd Warshall Algorithm.

Is this really an uninformed search problem?

Each node only has an id and no other information.

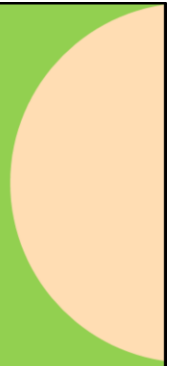
But ... what is this?



Could these be coordinates? Could they be helpful?

Lastly, it is true that our dataset did not include any information to base informed search algorithms on. But ... in our notebook we used an algorithm to visualize the graph. These algorithms produce a visualization by assigning coordinates to the nodes. 2-dimensional coordinates make for a good heuristic (based on euclidean distances). This may enable us to improve the performance of Dijkstra and apply other informed algorithms, which tend to gravitate to the goal node a lot faster.

We will explore this next time...



Questions?