PATH FINDING AND GRAPH TRAVERSAL

PATH FINDING AND GRAPH TRAVERSAL

- **Path finding** refers to determining the shortest path between two vertices i a graph.
- We discussed the Floyd–Warshall algorithm previously, but you may achieve similar results with the Dijkstra or Bellman-Ford algorithms.
- **Graph traversal** refers to the problem of visiting a set of nodes in a graph i a certain way.
- You can use depth- or breadth-first search algorithms to traverse an entire graph starting from a given "root" (source vertex).

PATH FINDING AND GRAPH TRAVERSAL

- Perhaps more useful to you are methods for finding traversable paths betwee multiple points.
- You have probably looked for such algorithms already. Today we will discuss one possible approach, namely the **A* (A star) search algorithm**.
- Key idea: Find the least-cost path from a source to one destination, out of on or more possible ones.
- Effectively builds a tree of partial paths, whose leaves are stored in a priority queue.
- Vertices are ordered in the queue according to a cost that consists of the distance travelled from source and an estimate of the distance to a goal (destination).

A* SEARCH

- Formally, the cost function for a vertex *n* is expressed as:

  f(n) = g(n) + h(n).

- *g(n)* is the cost of getting from the source to *n* and is tracked by the algorithn
- *h(n)* is a heuristic that estimates the cost from *n* to any possible goal.
- *h(n)* must be admissible, that is it should never overestimate the cost of reaching a goal (destination).
- One possibility is to compute *h(n)* as the shortest distance between any two vertices.

A* SEARCH

Operation:

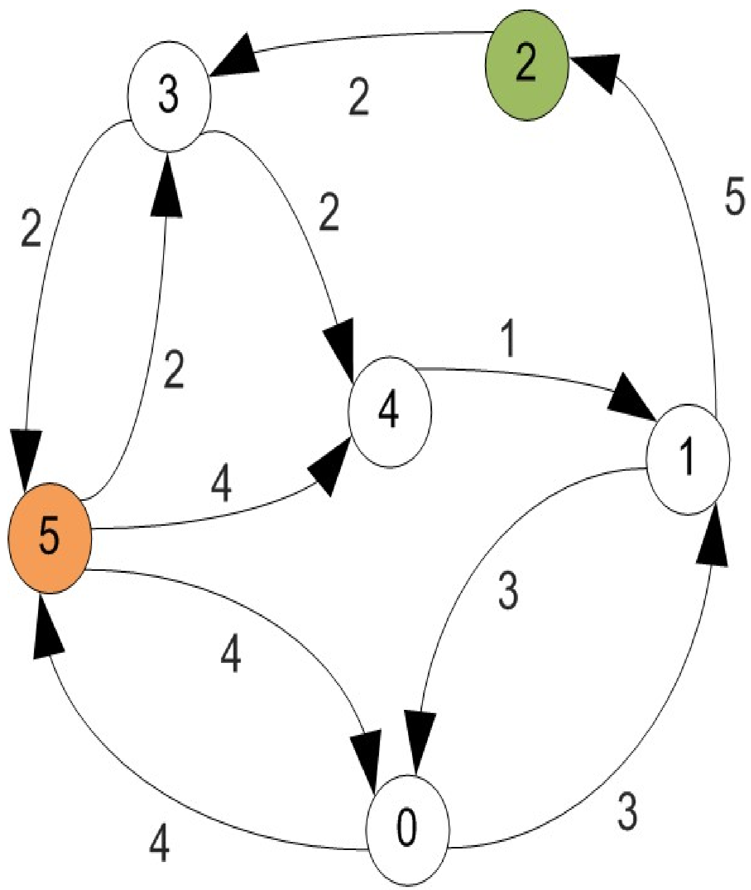
- First search routes most likely to lead to destination.
- This goes beyond best-first search, as it also considers the distance travelled so far.
- Maintain a priority queue of traversed vertices (open set/fringe). Low $f(n)$ corresponds to high priority.
- At each step, remove node with lowest f from queue, update the f (and g) values of their neighbours, and add neighbours to the queue.
- Stop when a goal node has a value f lower than any other nodes in the queue, or queue is empty.

A* SEARCH

Important remarks:

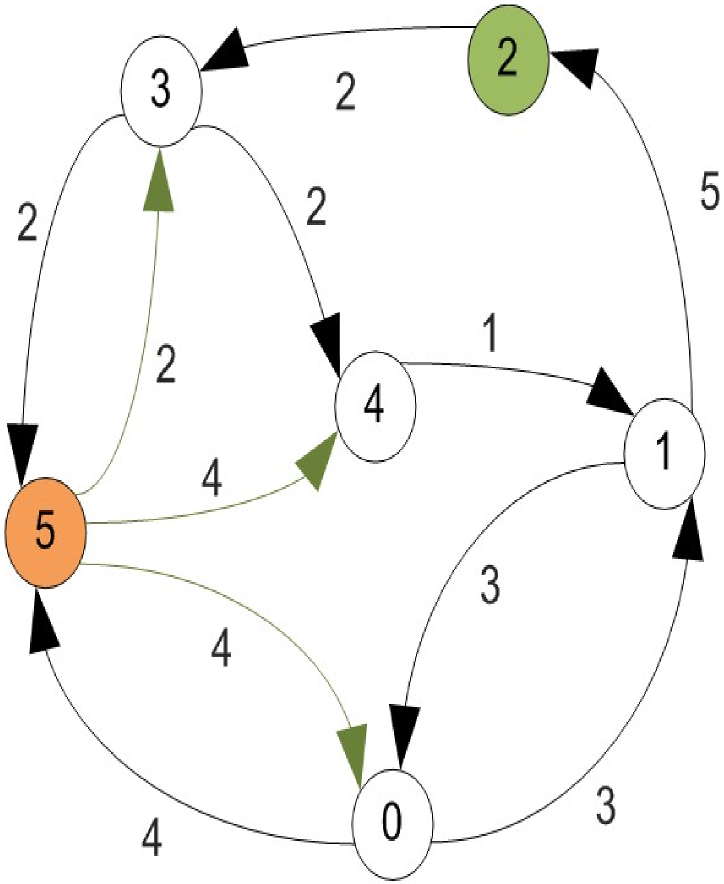
- Remember to set $g(n) = 0$ when n is a goal.
- To reconstruct the path, each node must keep track of its predecessor.
- You may want to perform some pre-processing on the graph at the start of th simulation (e.g. build the equivalent complete graph), to improve efficiency.

EXAMPLE



5, f = g+h = 0+1 = 1

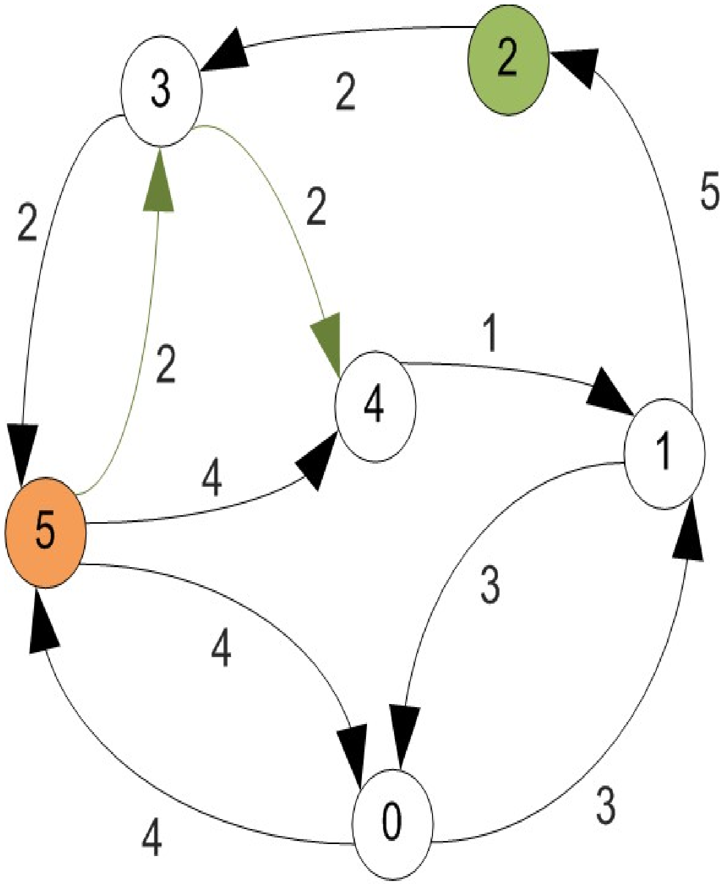
EXAMPLE



5, f = g+h = **0**+1 = 1

5-0, f = (0+4)+1 = 5   5-3, f = (0+2)+1 = 3   5-4, f = (0+4)+1 = 5

EXAMPLE

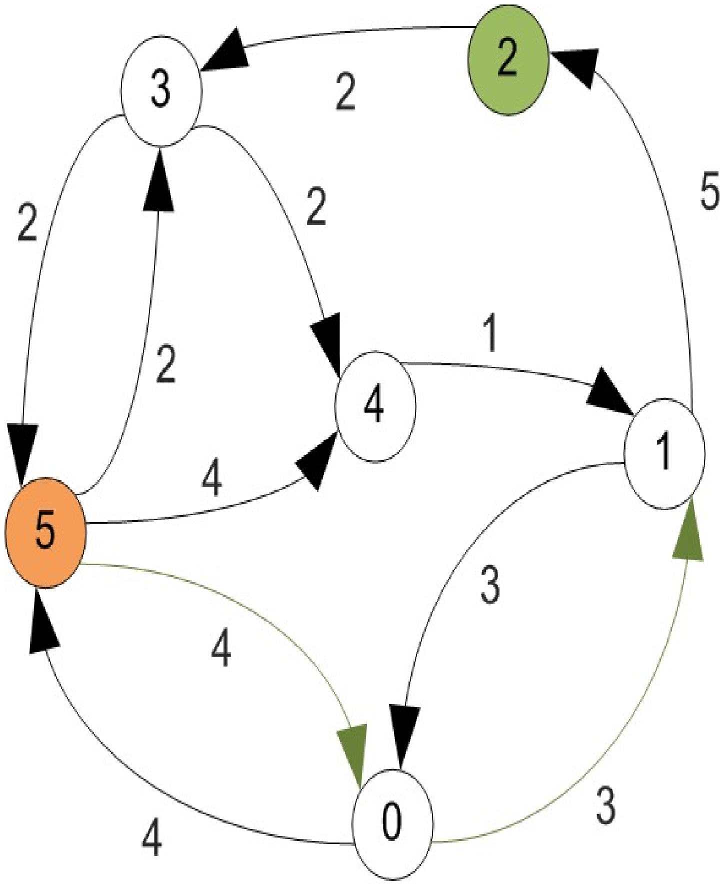


5, f = g+h = **0**+1 = 1

5-0, f = (0+4)+1 = 5   5-3, f = (0+2)+1 = 3   5-4, f = (0+4)+1 = 5

5-3-4, f = (2+2)+1 = 5

EXAMPLE



5, f = g+h = 0+1 = 1

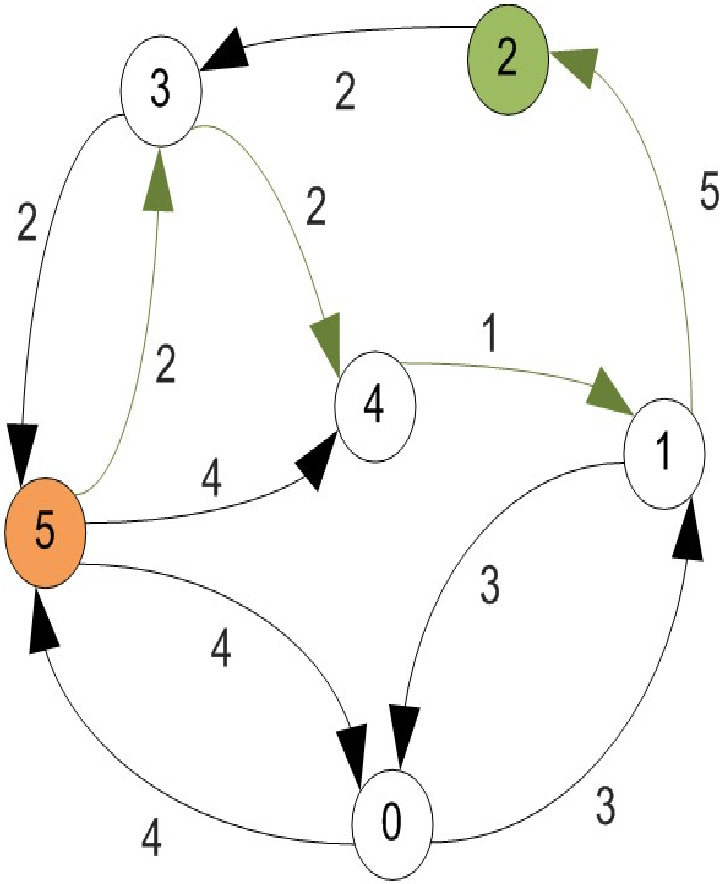
5-0, f = (0+4)+1 = **5** 5-3, f = (0+2)+1 = 3 5-4, f = (0+4)+1 = **5**

(alphabetical tie break)

5-3-4, f = (2+2)+1 = **5**

5-0-1, f = (4+3)+1 = 8

EXAMPLE



5, f = g+h = 0+1 = 1

5-0, f = (0+4)+1 = 5

5-3, f = (0+2)+1 = 3

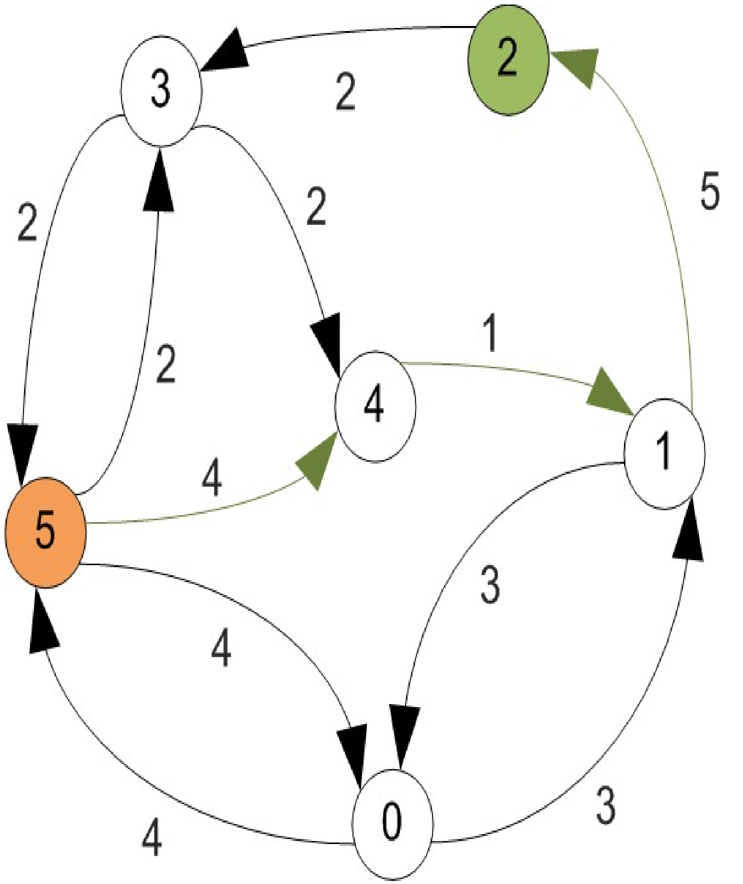
5-4, f = (0+4)+1 = 5

5-3-4, f = (2+2)+1 = 5

5-0-1, f = (4+3)+1 = 8

5-3-4-1, f = (4+1)+1 = 6

EXAMPLE



5, f = g+h = 0+1 = 1

5-0, f = (0+4)+1 = 5

5-3, f = (0+2)+1 = 3

5-4, f = (0+4)+1 = 5

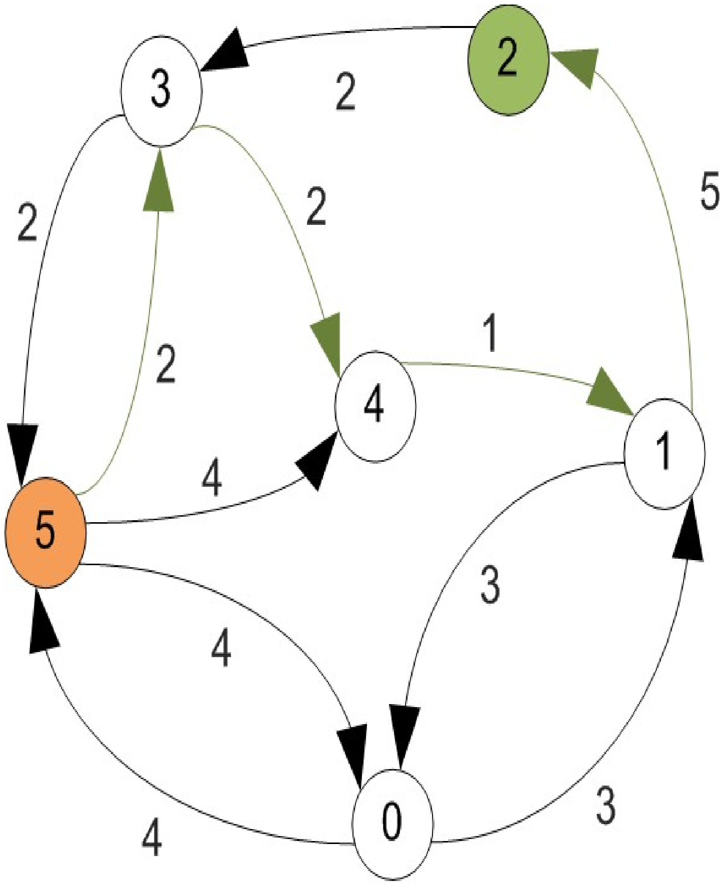
5-0-1, f = (4+3)+1 = 8

5-3-4, f = (2+2)+1 = 5

5-3-4-1, f = (4+1)+1 = 6

5-4-1, f = (4+1)+1 = <u>6</u>

EXAMPLE



5, f = g+h = 0+1 = 1

5-0, f = (0+4)+1 = 5

5-3, f = (0+2)+1 = 3

5-4, f = (0+4)+1 = 5

5-0-1, f = (4+3)+1 = 8

5-3-4, f = (2+2)+1 = 5

5-4-1, f = (4+1)+1 = 6

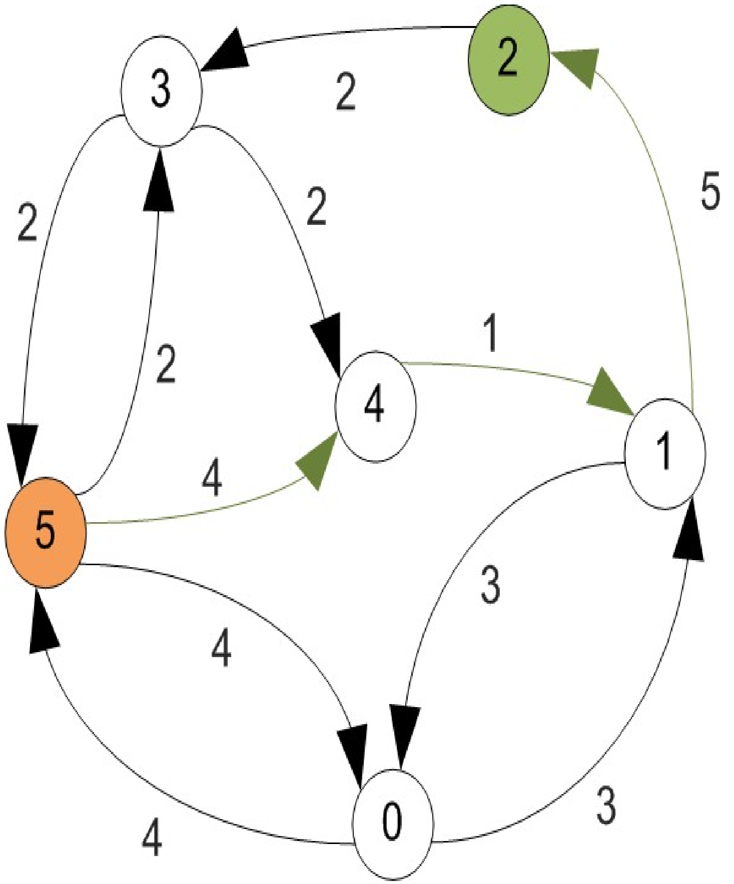
5-3-4-1, f = (4+1)+1 = 6

5-3-4-1-0, f = (5+3)+1 = <u>9</u>

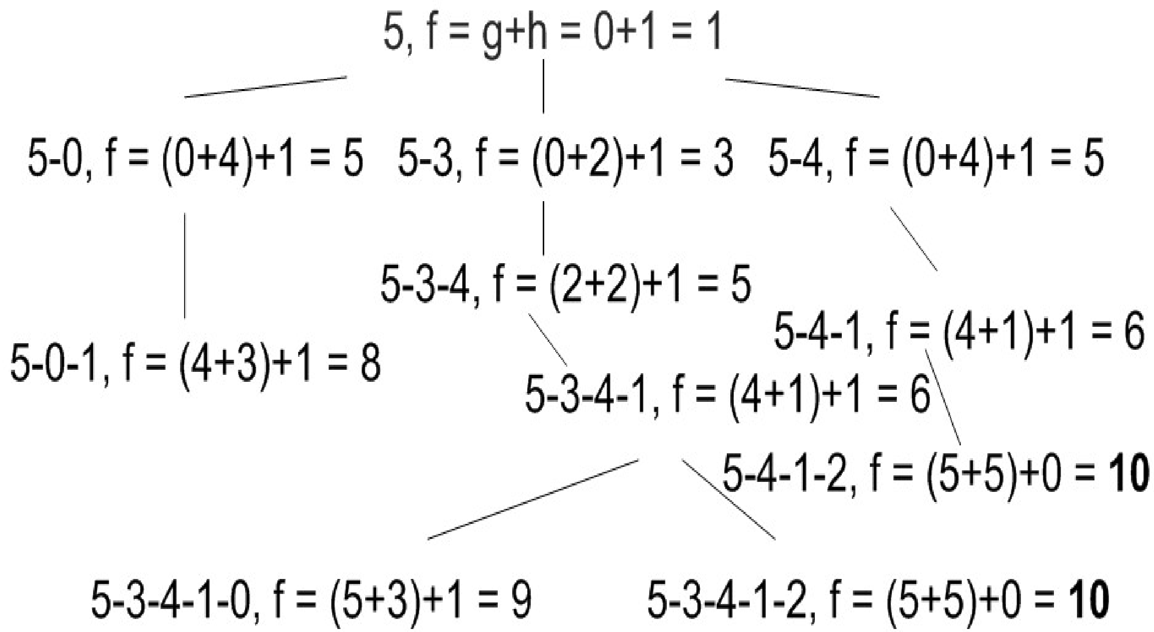
5-3-4-1-2, f = (5+5)+0 = **10**

(reached goal but did not finish expanding all branches)

EXAMPLE

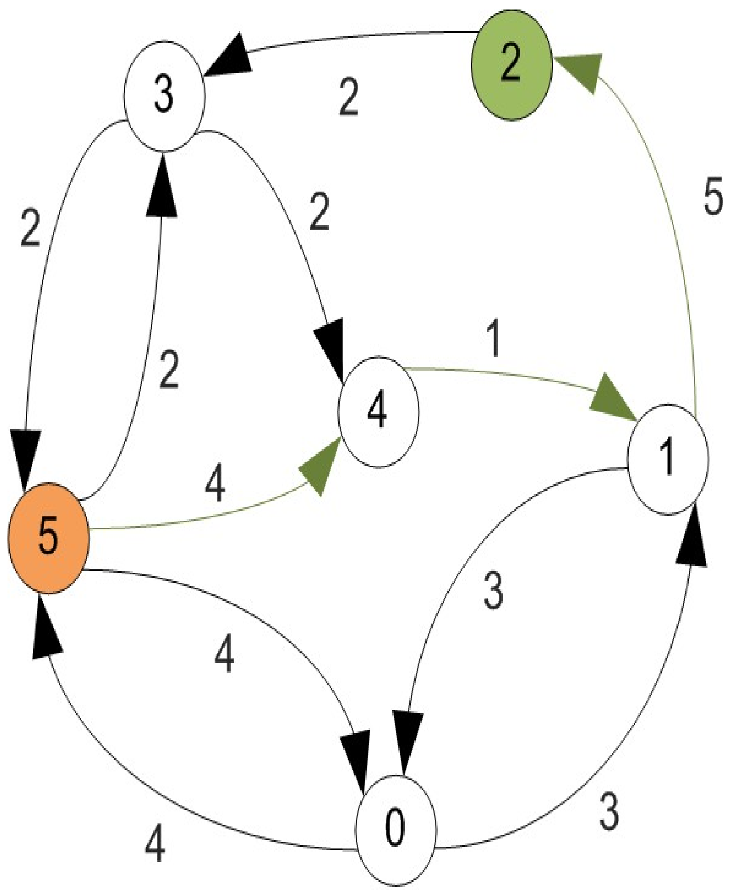


3
2
2
5
2
2
1
2
4
4
1
5
3
4
3
4
0



5, f = g+h = 0+1 = 1
5-0, f = (0+4)+1 = 5
5-3, f = (0+2)+1 = 3
5-4, f = (0+4)+1 = 5
5-3-4, f = (2+2)+1 = 5
5-4-1, f = (4+1)+1 = 6
5-0-1, f = (4+3)+1 = 8
5-3-4-1, f = (4+1)+1 = 6
5-4-1-2, f = (5+5)+0 = **10**
5-3-4-1-0, f = (5+3)+1 = 9
5-3-4-1-2, f = (5+5)+0 = **10**

EXAMPLE



5, f = g+h = 0+1 = 1

5-0, f = (0+4)+1 = 5 5-3, f = (0+2)+1 = 3 5-4, f = (0+4)+1 = 5

5-3-4, f = (2+2)+1 = 5

5-4-1, f = (4+1)+1 = 6

5-0-1, f = (4+3)+1 = 8

5-3-4-1, f = (4+1)+1 = 6

5-0-1-2, f = (7+5)+0 = **12**

5-4-1-2, f = (5+5)+0 = 10

5-3-4-1-0, f = (5+3)+1 = 9

5-3-4-1-2, f = (5+5)+0 = **10**

(reached goal, but at higher cost; further expanding would return to already visited nodes)

A* SEARCH

Optimising the execution time is possible by working with a so called *closed list*

- Start with an empty list.
- Add the nodes that have been expanded to the closed list.
- Do not explore other branches of a node that is already in the closed list.

PSEUDO-CODE

```
create closed list; initialise empty
create open list of vertices we currently work on; initialise with source

while (goal not reached) {
   consider best node in open list (lowest f value)
   if (goal reached) {
      done, return;
   } else {
      move current node to closed list and examine all its neighbours
      for (each neighbour) {
         if (in closed list current g value is lower) {
            update neighbour with the new, lower, g value
            change neighbour's parent to current node
         } else if (neighbour in open list and current g value is lower) {
            update neighbour with the new, lower, g value
            change neighbour's parent to current node
         } else {  // neighbour in neither open or closed list
              add neighbour to open list and set its g value
         }
      }
   }
}
```

REVIEW OF PART 1

SUBMISSIONS

- 36 out of 82 students submitted at least something.
- That's 44% submission rate –was expecting something slightly above 50%.
- Many of you did not ask any explicit questions nor highlighted any aspects on which you wanted feedback.
- A couple of submissions did not compile!

MULTIPLE FILES

- There seems to be a preference for using a relatively large number of files, given the small size of this project.
- There is nothing wrong with that, but pay attention to memory management.
- Using more/less files will not be penalised – just explain why you chose to implement things in a certain way.
- If refactoring is a reason, do emphasize that.

THE READMES

- Ranged from very basic ones, containing a couple of lines, to very detailed ones.
- Most of them explained how to build and execute the simulator, which is good. For Part 2 submit a makefile.
- Some acknowledged the limitations of the code and problems known at the time of submission.
- Some did not state explicitly at which stage of the development they were.
- Some did not submit a README – you will be losing points if this happens for Part 2.

RANDOM GOODNESS

- *“I was pushing the submission to the last possible moment, in (vain) hope that I will be able to debug the program.”*

RANDOM NOT-SO-GOODNESS

- Low-level coding decisions:
  - Prone to change and you *will* forget to update the README
  - Should be in comments in the source code file concerned – some code lacked commenting altogether.
- High-level structure: you are more likely to remember to change the README in case of a major re-structuring.

RANDOM NOT-SO-GOODNESS

- Make sure you read carefully the requirements
  - On one occasion the first command line parameter was not the input script, but a keyword that introduced the input script.
  - Although working to some extent, some did not implement any command line parsing at all.
  - One submission was prompting the user for input after the execution started – remember that your code will be automatically tested.

REFACTORING

- Only a small number of READMEs contained mentions of refactoring,
- Both with reference to future refactoring:
  - Either promising to refactor later **or**
  - Done a good bit of refactoring already, but planning to do more.
- It was still early days; however, refactoring is something you should be *trying* to do constantly.

INVALID INPUT

- Try to think of exceptions which you really do not believe can happen under normal executing conditions.
- The user may simply make some typing errors when producing the input.
- Some parameters may have been given in an order different than the expecte one.
- When is this a serious problem?
- Distinguish between warnings/errors where possible.

INVALID INPUT

- What should you do if you discover you have incomplete information during simulation run?
- For example, you attempt to retrieve the `noBuses` and find that this was not given.
- This is not problematic for parsing, because the user may have simply forgotten to specify the fleet size.
- However, if you validate the input before running the simulation, then it real becomes a problem to find you have no buses to schedule during the simulation.
- The simulation should not have been started since the validation should have uncovered the error.

FURTHER CHECKS

- Check the sign of numbers.
- Check whether it is indeed numbers you find when numbers are expected.
- Check if a number has the type you expect (integer/real).
- Be careful with hours/minutes/seconds conversions.

INPUT SCRIPTS

- You have not spend much time authoring input scripts.
- I recently made some examples available, but it is essential that you produce your own test scripts.
- Network size is not the only thing that matters.
- Think of scenarios where e.g. route planning algorithms may have a hard time.

FINAL ADVICE

FINAL ADVICE (I)

- I said this already, but read the handout carefully again
- Submit a README file and explain your design choices and main building blocks therein, as well as any known bugs, limitations, and requirements not implemented or additional features implemented.
- Include a Makefile. Running 'make' in the *root directory of your project* should be enough to build an executable.
- Make sure you expect an input script as a command line parameter and chec[k] that the one specified by the user actually exists.

FINAL ADVICE (II)

- Do not prompt the user for input.
- Implement a parser. If you do not manage to get this working, not much can be tested.
- Conform to the output specification (remember the automated testing).
- Build a set of tests and submit them as evidence of testing. Explain briefly what you tested for with each.
- Comment your code, though not excessively.
- Clean up your code, have consistent spacing, remove commented out blocks.

IN SUMMARY...

- Strive at least for a half decent simulator.
- Make sure it compiles and runs on DiCE, and it does not require super-user privileges!
- As a backup plan, have at least a planning algorithm that works in a taxi fashion (one bus allocated to each request, if possible).
- There is no minimum page requirement for the report: "Demonstrate you are familiar with discrete-event stochastic simulations, you can interpret the output, understand the system's performance and you can present your results in a clear and concise manner."