Path Finder

CMPUT 229

University of Alberta

Your task in the lab

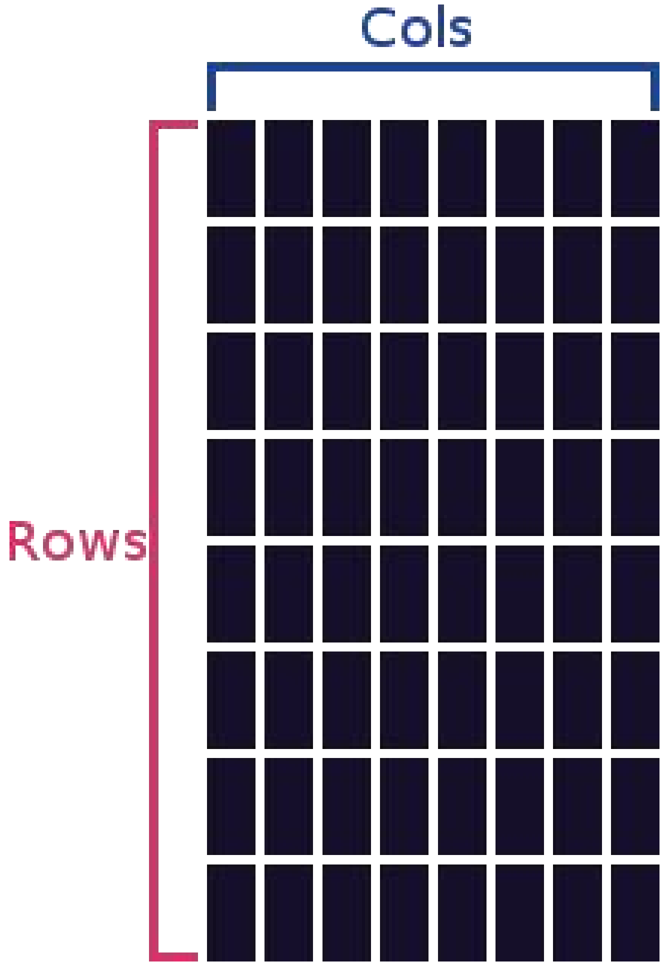
- Implement A* search in RISC-V
- Create a search visualizer for A* search in RISC-V with the help of GLIR (Graphics Library for RISC-V)

GLIR

- **<u>G</u>**raphics **<u>Li</u>**brary for **<u>R</u>**ISC-V.
- GLIR is a library built at the University of Alberta.
- It has a collection of subroutines to emulate graphics.
- It prints graphical shapes onto the terminal.
- GLIR contains functions to print lines, rectangles, triangles, and circles.

GLIR: Terminal

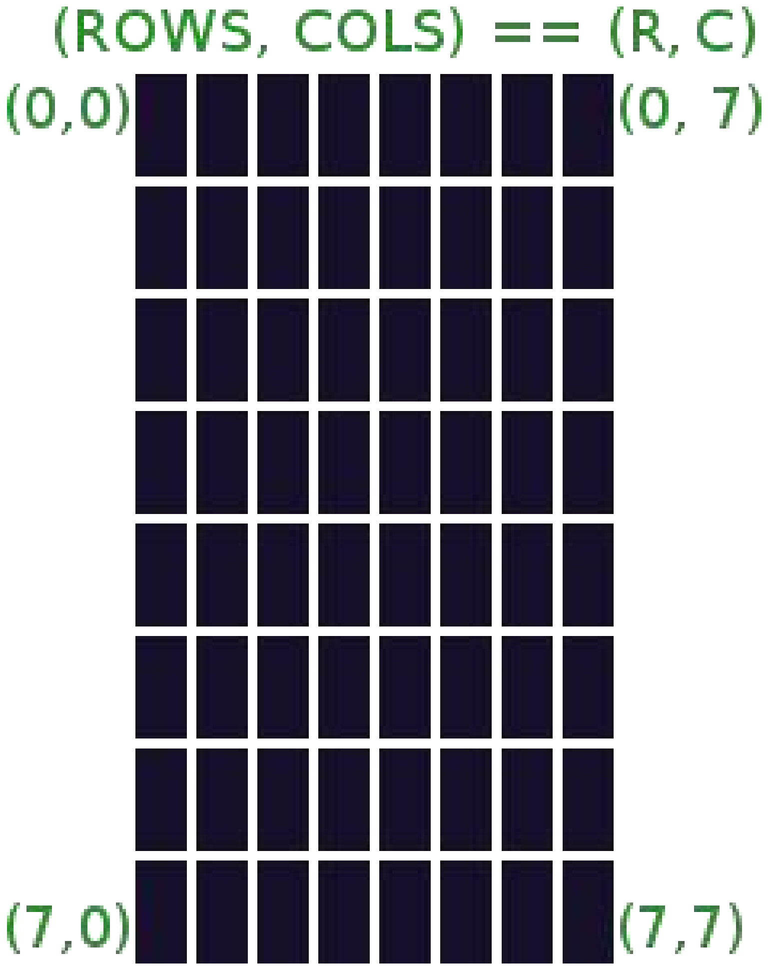
- The terminal is where the graphics will be rendered.
- Grid of rectangular cells making up rows and columns.
- Each of these cells can have a character, a background colour, and a foreground colour.



source: GLIR

GLIR: Terminal (cont'd)

- Rows and columns describe the position of a cell.
- Similar to the Cartesian coordinate system.
- But the tuple for a cell on the cell is (Row, Col), not (Col, Row).
    - In other words, it uses the form (y, x) rather than the usual (x, y) in the Cartesian coordinate system.
- This is because terminals were designed to print text top to bottom, left to right.
- This is also why the origin (0, 0) is at the top left of the terminal.



source: GLIR

GLIR: Preparation and Cleanup

- `common.s` calls `GLIR_Start` and `GLIR_End` before and after the visualizer process.
- These are two important procedures in GLIR.
- **`GLIR_Start`** (preparation):
  - Resizes the screen to the user-specified size.
  - Hides the terminal cursor.
  - Clears the terminal to the default background color.
- **`GLIR_End`** (cleanup):
  - Resizes the screen back to default (24x80).
  - Shows the terminal cursor.
  - Clears all the previous terminal output.

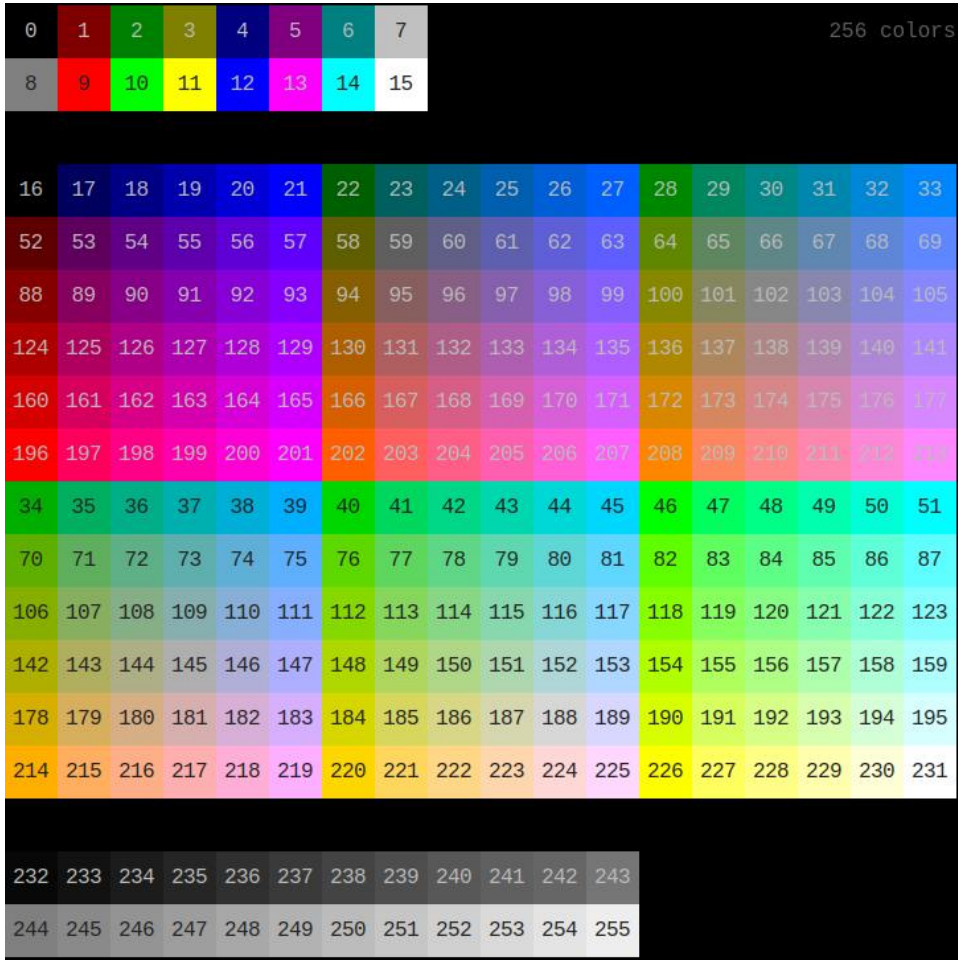
`common.s:`

```
…

jal     GLIR_Start

# Run the visualizer
jal     pathFinder

# End the GLIR terminal
jal     GLIR_End

…
```

GLIR: Color

- GLIR supports 256-color terminals.
- It changes colors using ANSI escape codes.
- ANSI escape codes are a set of codes that can be used to change terminal options such as cursor location, font styling, and colors.
- GLIR abstracts away these ANSI escape codes to allow the user to simply pass it the desired color code from the Xterm 256 colors.
- The list of Xterm 256 colors can be found here: https://www.ditig.com/256-colors-cheat-sheet

GLIR: Color Table



Environment

- $m \times n$ eqaully sized cells
- From any cell:
  - cannot move off the map
  - cannot move into a water cell
  - can only move into adjacent cells (cells immediately on the left, right, top, and bottom)
- Number each cell with a unique non-negative integer
  - The most upper left cell is numbered 0
  - Increment by one each time we move one cell to the right.
  - If we reach the end of the row, we wrap to the left most cell one row below, and continue

Example

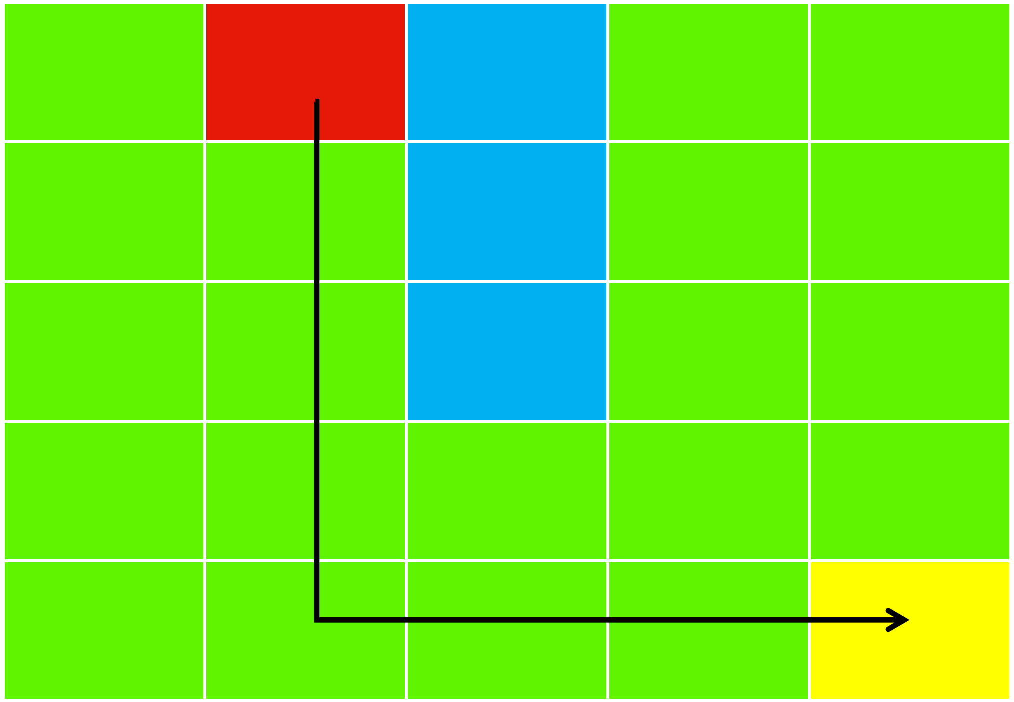
From cell 1:

- Can move into cells 0, and 6
- Cannot move up
  - move off the map
- Cannot move into cell 2
  - water cell
- Cannot move into cell 5
  - diagonal from cell 1

Paths

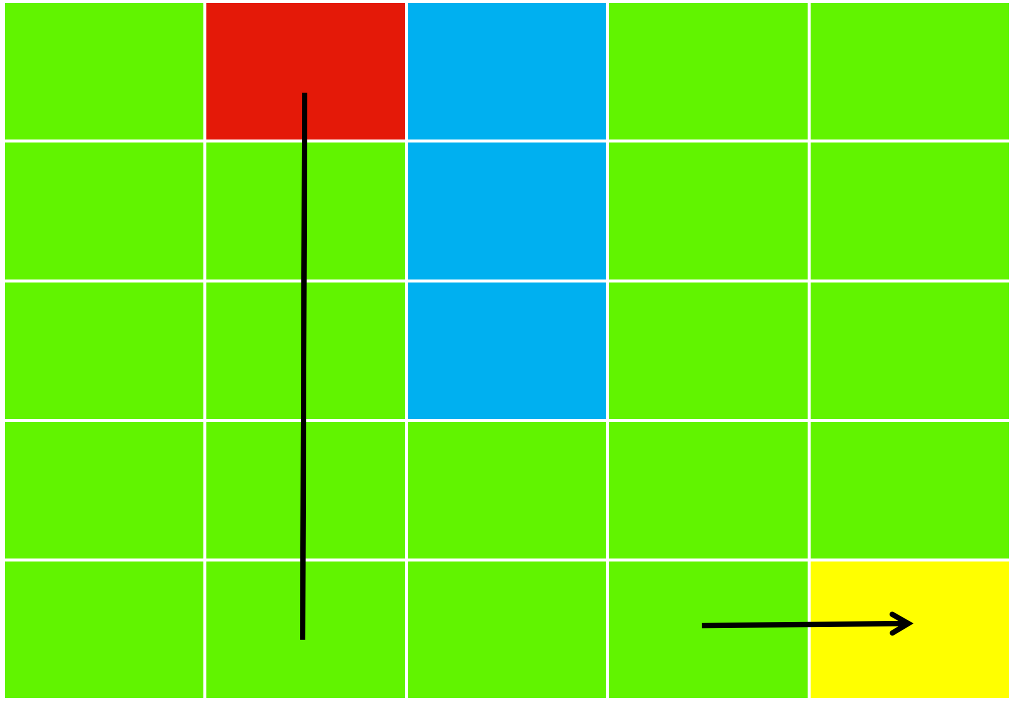
- Defined between two cells
- Must obey environmental constraints

Valid Path



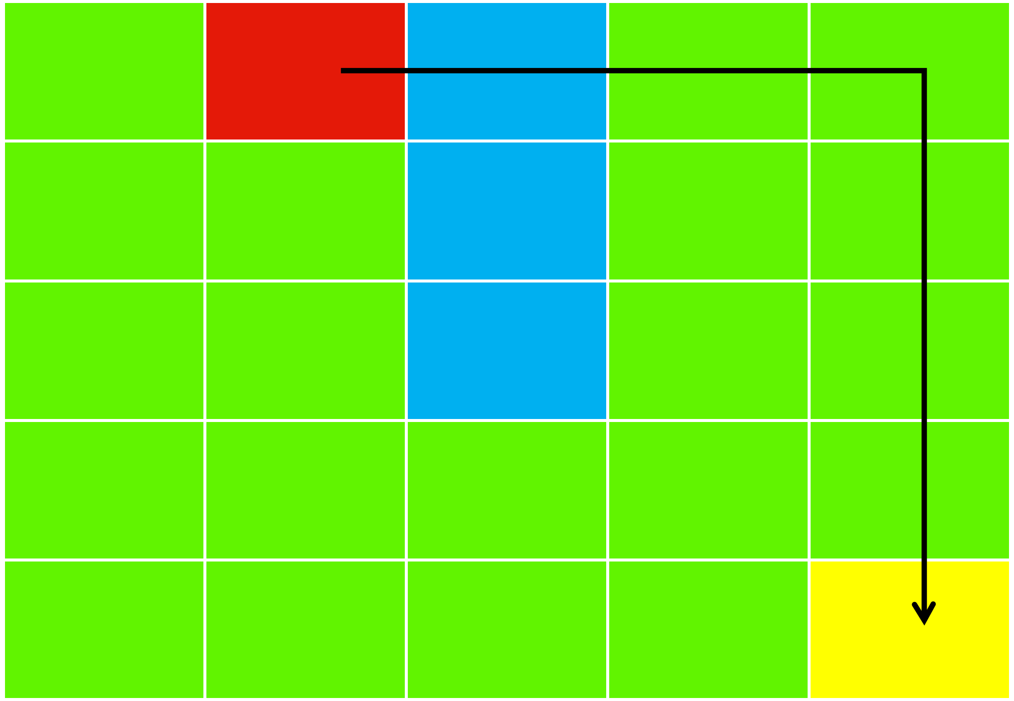
- Contiguous
- Consists of only grass cells

Invalid path



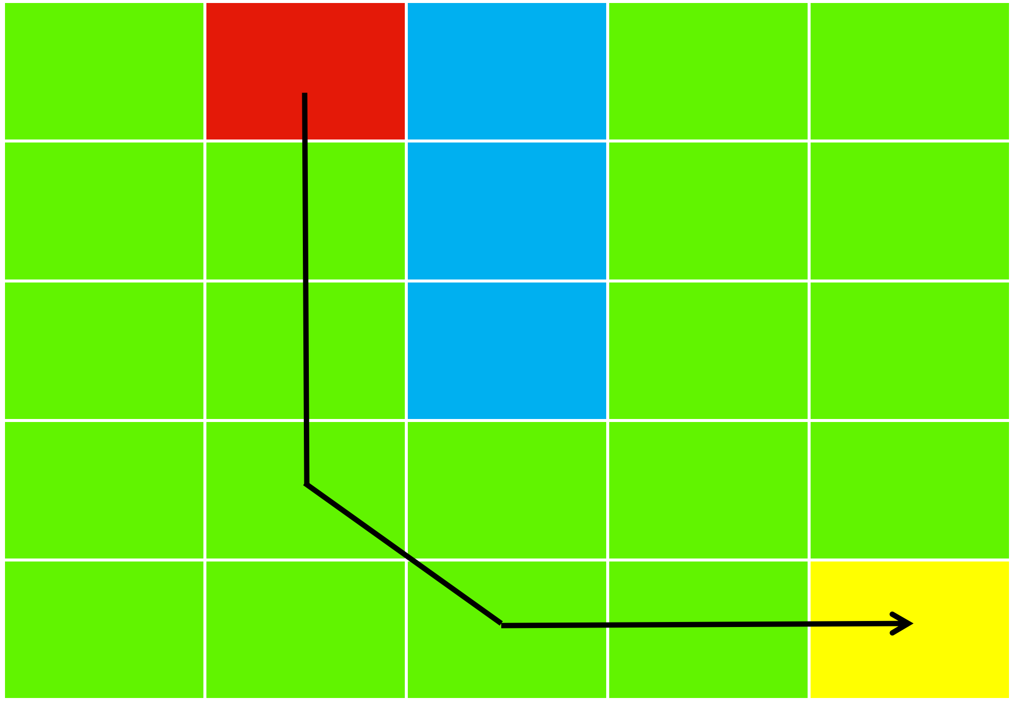
- Path is broken
- A* cannot jump over cells

Invalid path



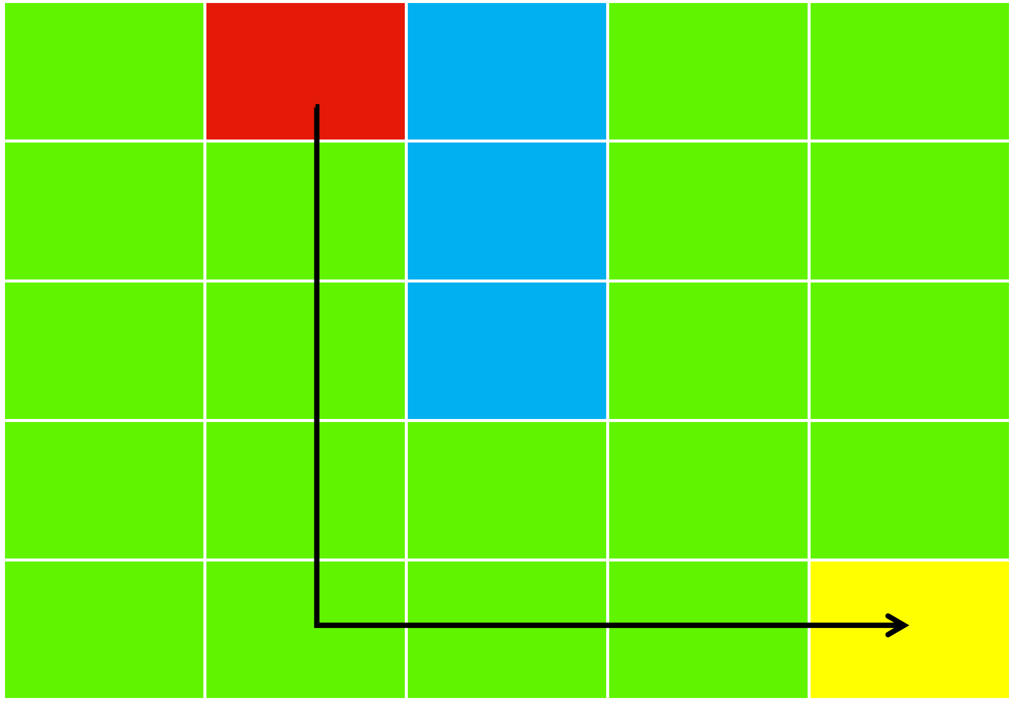
- Path crosses a water cell
- A* cannot move into water cells

Invalid path



- Diagonal pathing
- A* cannot move diagonally from one cell to another cell

Path Concepts - Representation



- Represent a valid path as an array of cell numbers.
- The path on the left can be represented as:
  1, 6, 11, 16, 21, 22, 23, 24

Path starts at cell 1

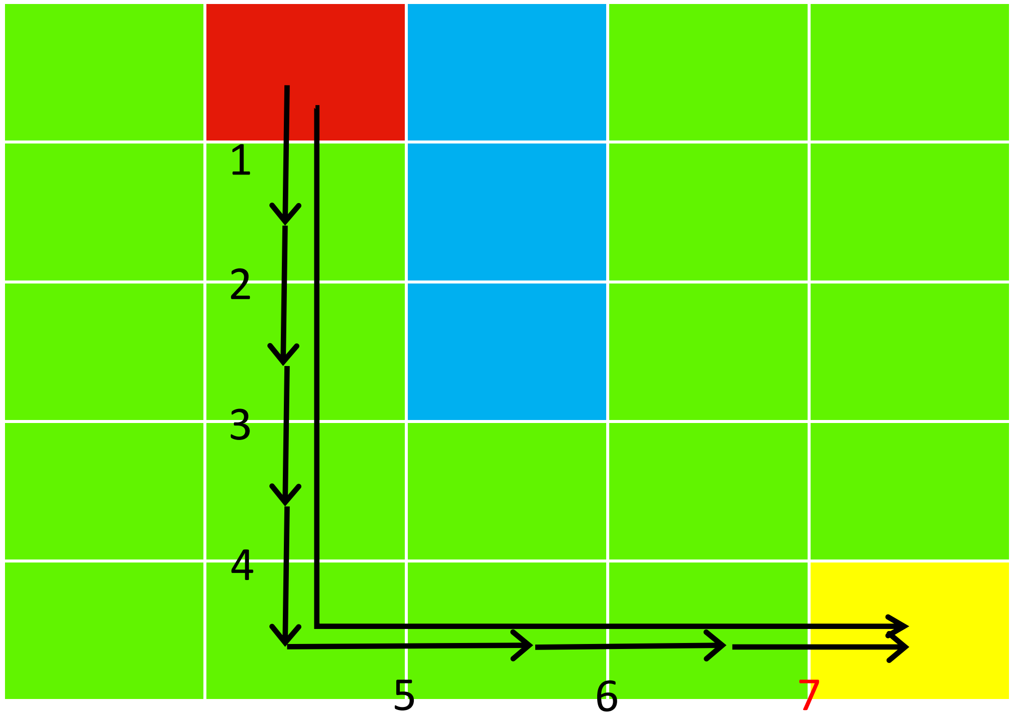
goes through cells 6, 11, 16, 21, 22, 23 (in that order)

and terminates at cell 24

Path Concepts - Distance

- Distance between adjacent cells is one (1) unit
- Distance of a path is the distance between the first cell and the last cell of the path
- For a path with *n* cells, traveling from the first cell to the last cell on the path takes *n - 1* moves.
- Therefore, the distance for a path with *n* cells is *n - 1* units.

Path Concepts - Distance



- Path contains eight cells: 1, 6, 11, 16, 21, 22, 23, 24
- Distance is 7 units

A* - Introduction

- A pathfinding algorithm
- Uses heuristic functions to estimate the distance to the goal.
- By using heuristic function that never overestimate distances, A*...
  - is guaranteed to find the shortest path, if it exists;
  - saves time and memory by prioritizing search on seemingly shorter paths

A* - Terminologies

- For a particular valid path, *P*, from the start to an arbitrary cell, A, the **`parent`** of *A* is another cell, B, that comes immediately before *A* on *P*
  - The parent of the start cell is defined to be itself
  - Accordingly, cell *A* is a **child** of cell *B*
- For a particular valid path, *P*, from the start to an arbitrary cell, A, the **g** of *A* is the distance of *P*
- **h**: Estimated distance *A* to the goal
- **`f`** `:= g + h` (**`f`** is defined to be `g + h`)

`parent` and `g` - An Example

Consider the path 1, 6, 11, 16, 21, 22, 23, 24

The `parent` of cell 24 is cell 23

- Cell 23 comes immediately before cell 24 on this path

The `g` of cell 24 for this path is 7 units

- The distance of this path is 7 units

A* - Terminologies

Uses two lists to store information:

1. Closed List
    - Stores relevant information about each cell
2. Open List
    - Keeps track of the cells that are not explored yet

A* - Terminologies

- Visit a cell ≡ Record the **`parent`** and **g** of the cell
- Expand a cell ≡ Visit its adjacent cells

A* - Algorithm

Search begins with the start cell

1. Visit and expand the start cell
2. Repeatedly expands visited cells until...
   - A* expands the goal → a solution is found
   - No more visited cells to expand → no solutions found

A* - Algorithm

Uses two techiques to find the shortest path

1. Expands the cell with the smallest **`f`** first
2. A* keeps the **`parent`** and **`g`** of a cell *A* only for the shortest path from the start to *A*

A* Pseudocode

- The webpage contains the pseudocode for A*

Pathfinder

- There are many pathfinding algorithms
- Pathfinding visualizers graphically shows how different pathfinding algorithms search the environment for the shortest path from the start to the goal
- Examples:
  - https://pathfindout.com/

Pathfinder Implementation

We implement a visualizer for:

- One algorithm: A*
- An environment with only two types of cells: grass and water

Four main components:

1. Map buffer
2. Water array
3. Closed list
4. Open list

Map Buffer

- Holds the internal representation of the map
- 1D array where the `i`'th element is a...
  - 1 if cell `i` is a water cell
  - 0 otherwise

Water Array

- An array of integers
- Each element is the cell number of a water cell on a particular map
- A pointer to the water array will be passed as an argument to the `pathFinder` function

Closed List

- An array of structs, one struct for each cell in the map
- Each struct contains three words in the following order
    1. **parent**
    2. **g**
    3. **h**
- The corresponding struct of cell `i` will be the `i`'th struct in the array
- A **parent** of `-1` indicates that the cell has not been **visited** yet
- Record **parent**, **g**, and **h** if cell was visited
- If A* finds a shorter path to a cell, update its **parent**, **g**, and **h**

In-Memory Representation

Map:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Legend:

Grass	Water	Start	Goal

Water Aarray:

Value	**2**	**7**	**12**
Index	0	1	2

Map Buffer:

Value	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Closed List:

parent → -1, g → 0, h → 0 (index 2)

Value	-1	0	0	1	0	7	-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0	...	-1	0	0
Index	0			1			2			3			4			5			6			...	24		

Open List

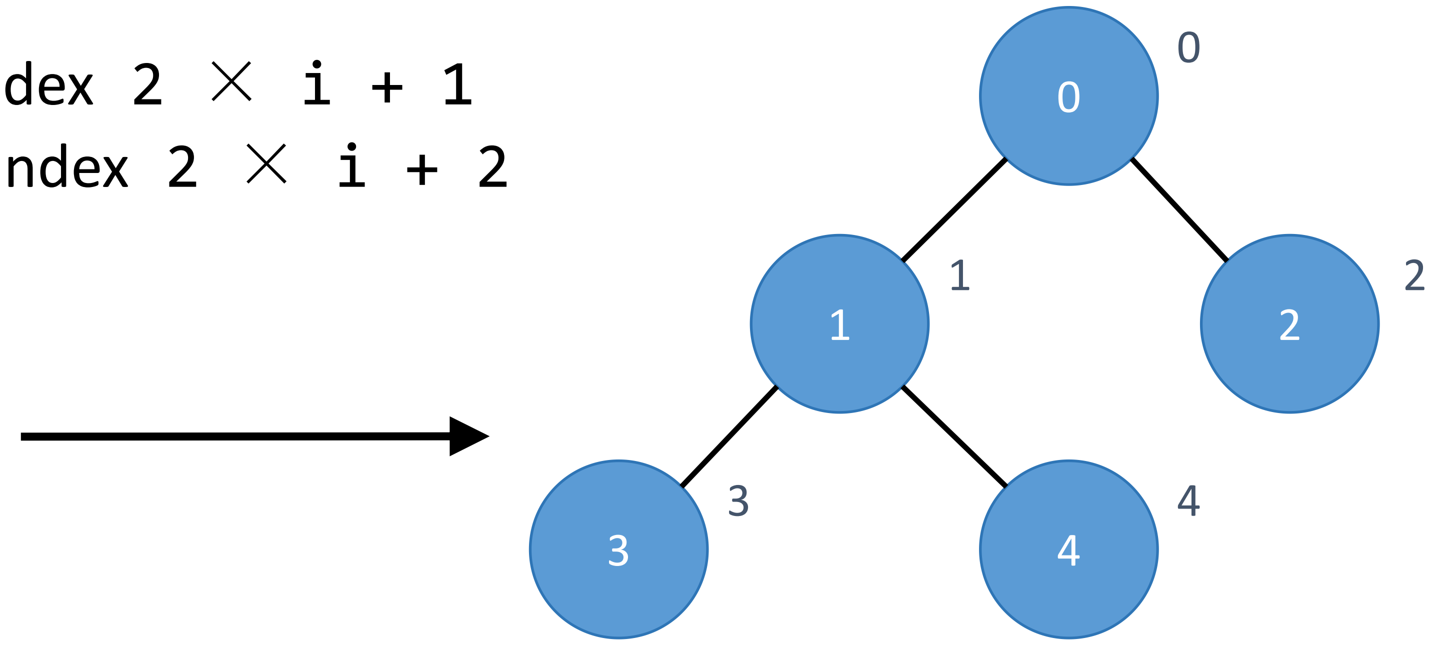
- Keeps track of the cells that are **visited** but **not expanded** yet.
  - To expand a cell, A* first remove it from the open list
- Contains only the cell number of the cells
- Cells are added and removed from the open list very frequently
- Need an efficient implementation - min-heap

(Min-)heap

- A complete binary tree that satisfies the heap property
- Implemented as a 1D array
  - Root has index 0
  - Left child of node `i` has index $2 \times i + 1$
  - Right child of node `i` has index $2 \times i + 2$

Value	0	1	2	3	4
Index	0	1	2	3	4

Array Representation



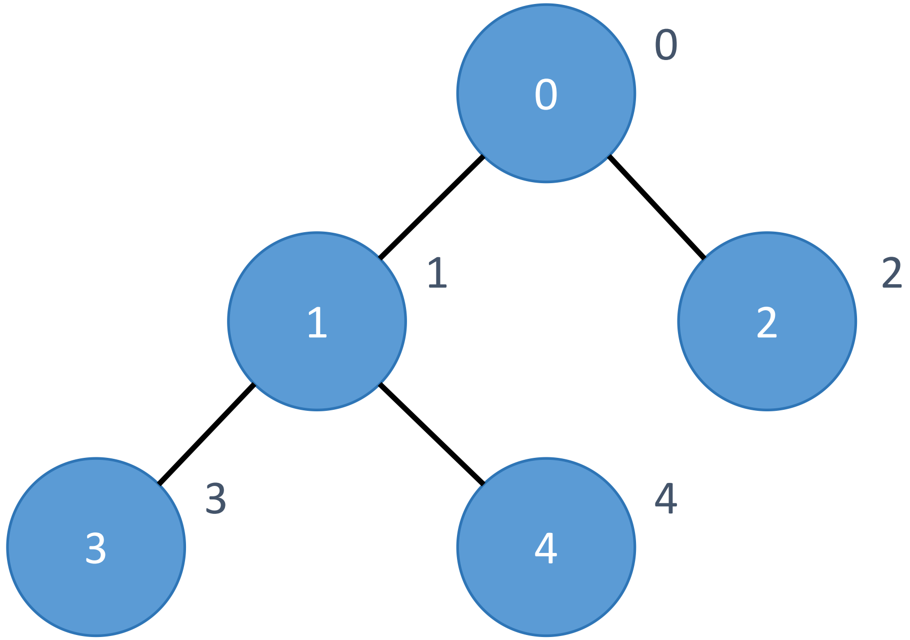
Tree Representation

Heap Property of Min-Heap

- The root node must have the **smallest** key
- For any given node, its key is **less than or equal** to the key of its children (if any)
- We will use the **f** value of each cell as the key
- Must be checked when inserting, deleting, or changing the key of an element
  - If the heap property no longer hold, elements must be re-arranged s.t. the heap property holds again

Heap Property - Example

Tree Representation



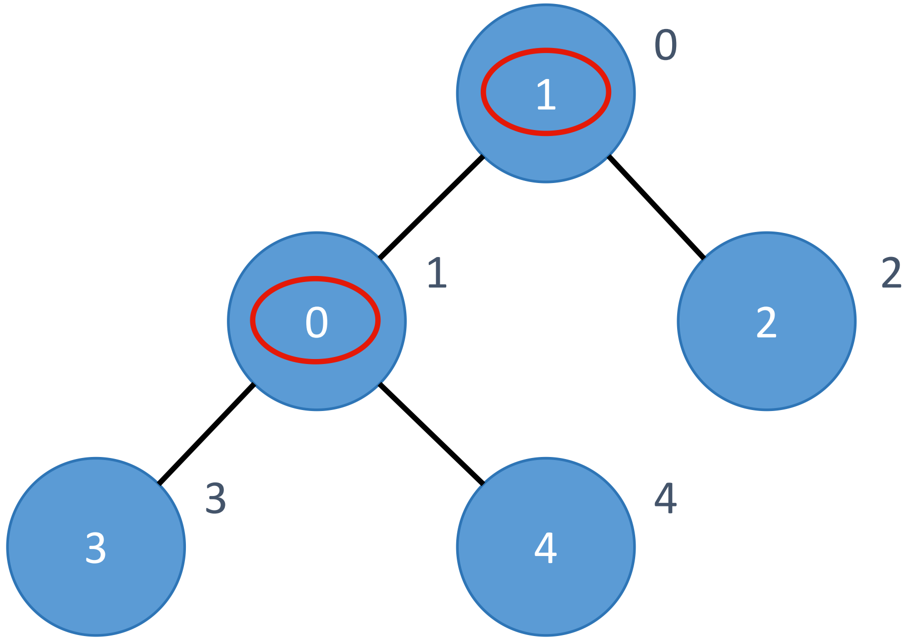
Array Representation

Value	0	1	2	3	4
Index	0	1	2	3	4

Satisfies the min-heap property

Heap Property - Example

Tree Representation



Array Representation

Value	1	0	2	3	4
Index	0	1	2	3	4

Does not satisfy the min-heap property

The key of the root node is greater than the key of its left child

Heap Operations

- This lab provides three heap operations in the `heapq.s` file
    1. `insert`: inserts a cell into the heap and maintains the heap property based on the **`f`** values of the cells
    2. `popMin`: removes the cell with the smallest **`f`** from the heap and maintains the heap property based on the **`f`** values of the cells
    3. `minHeap`: transforms an array of cell numbers into a heap in place based on the **`f`** values of the cells
- Specifications for the three functions are on the webpage

Heap - Notes

- Although having a high-level understanding of the heap data structure and the heap operations is sufficient to complete the lab...
- It is strongly recommended that students take a look at the source code in `heapq.s`

Initialization

- `common.s` declares the map buffer, closed list, and open list...
- ... and passes the pointers to each as arguments to the `pathFinder` function
- Students must initialize the arrays with initial values
  1. The map buffer is initialized as an arrays of zeros
  2. Each element in the closed list is initialized as `-1,0,0`
  3. To initialize the open list, simply set the size of the open list to zero
     - The size of the open list is given a global variable in the `heapq.s` file

Heuristic Function

- Each cell is associated with a coordinate `(R, C)`
- We can use this coordinate to calculate the Manhattan distance between two cells

Manhattan Distance

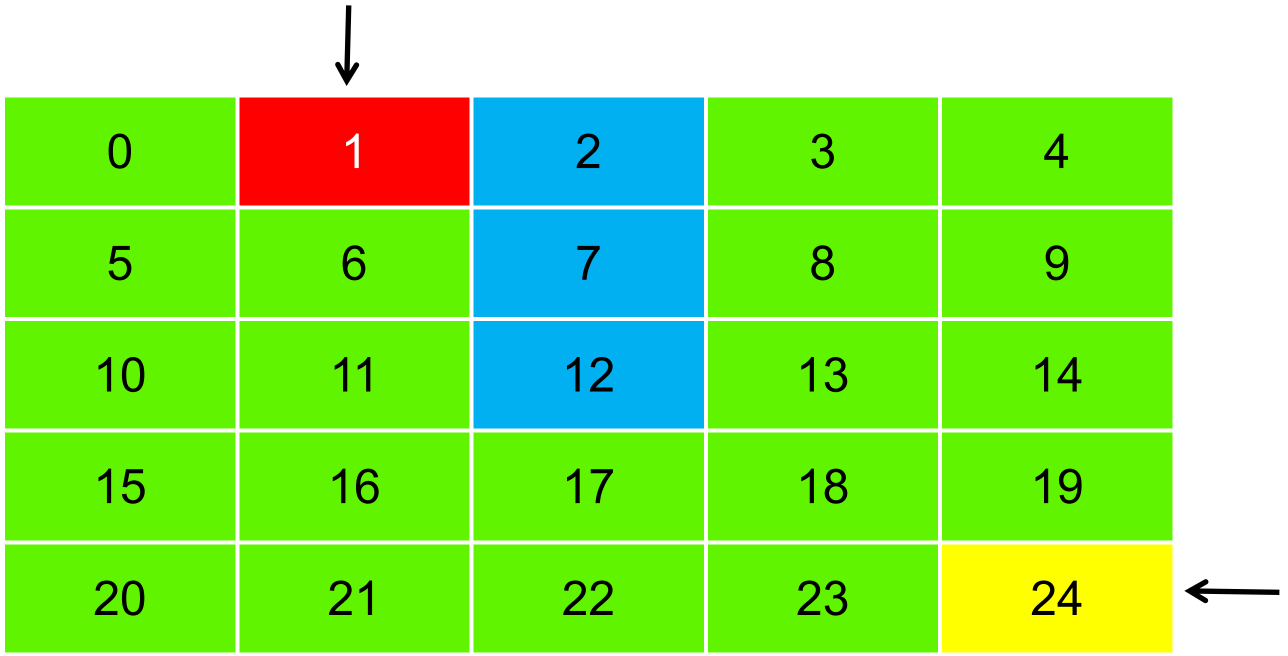
- The Manhattan distance between two cells with coordinates (R_1, C_1) and (R_2, C_2) is:

$$|R_1 - R_2| + |C_1 - C_2|$$

- The absolute difference between the row numbers plus the absolute difference between the column numbers

Manhattan Distance - Example

$(R_1,C_1) = (0,1)$



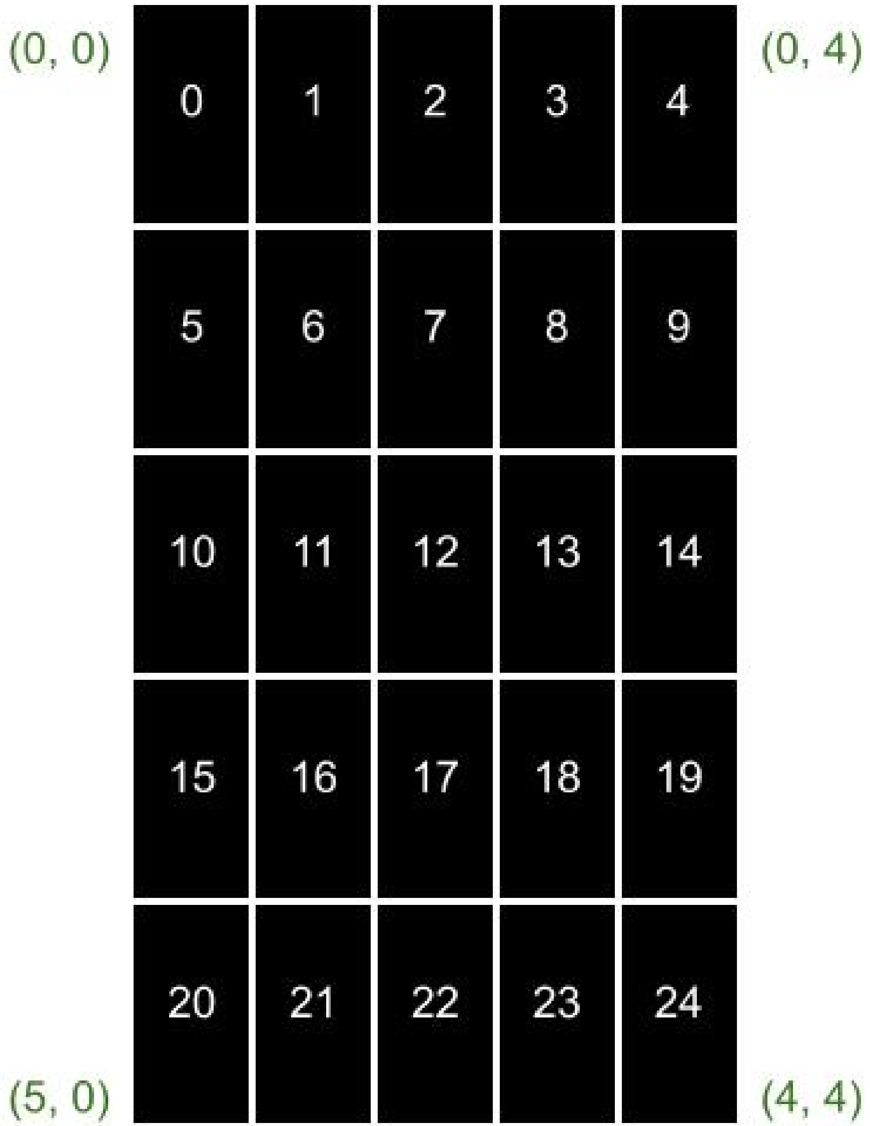
$(R_2,C_2) = (4,4)$

$$
\begin{aligned}
&|R_1-R_2|+|C_1-C_2| \\
&= |0-4|+|1-4| \\
&= |-4|+|-3| \\
&= 4 + 3 \\
&= 7
\end{aligned}
$$

Drawing the Map with GLIR

Align cell 0 with the cell located at (0, 0)

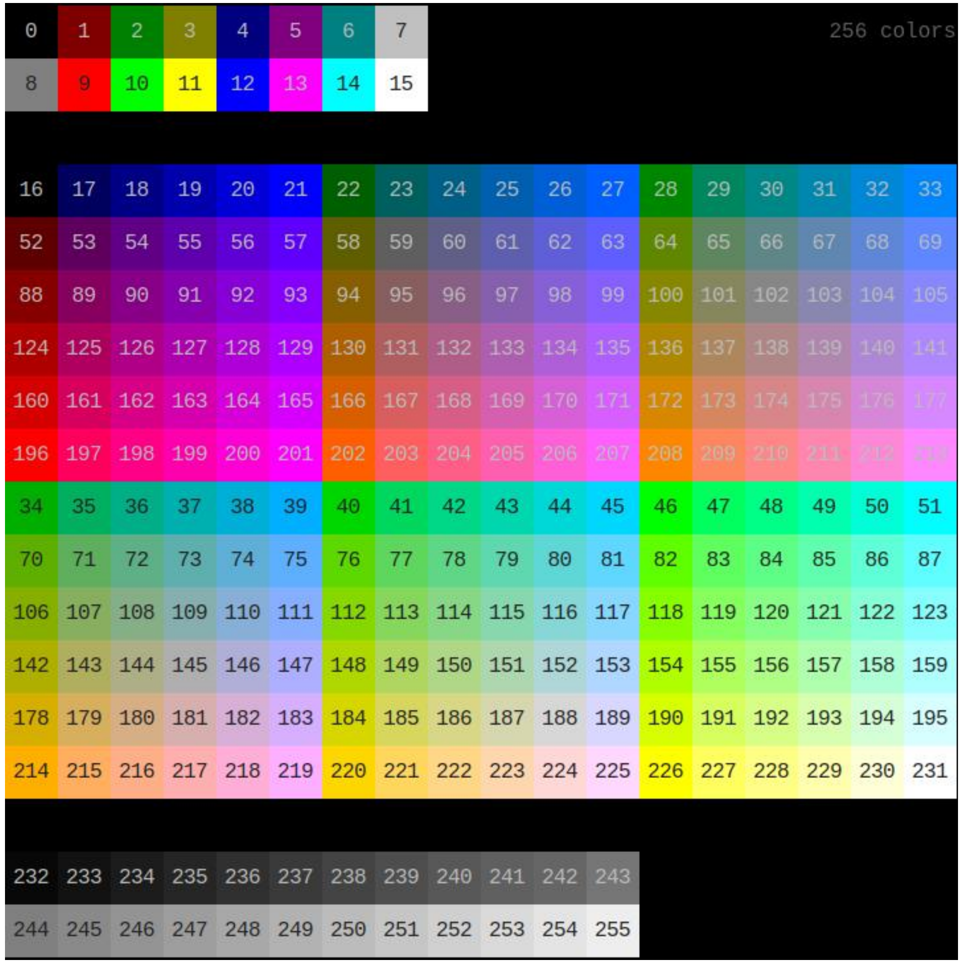
For example, the coordinate of cell 16 is (4, 1)



Drawing the Map - Colors

- Grass → 10
- Water → 14
- Start → 9
- Goal → 11
- Expanded cells → 8
- Solution path → 13

Color codes are given as global variables in the `common.s` file



Drawing the Map - Updates

- There are multiple ways to display screen updates.
- The GLIR documentation points out two methods:
    - Clear and Refresh
    - Batch and Release.
- These two methods are helpful to know, but they are not appropriate for this lab.
    - There will be a lot of screen updates, so the Clear and Refresh method will result in flickers because clearing and printing onto the screen is a relatively slow process.
    - For printing relatively simple shapes (one cell at a time) in this lab, using the Batch and Release method is excessive and unnecessary.

Drawing the Map - Updates

Instead, this lab uses the following method

1. Print the initial map to the terminal
2. Redraw cells in gray as A* expands them
3. If a solution path is found at the end, we redraw the cells on the solutin path with purple
4. Redraw the start and goal cells

All of the steps can be achieved using the `GLIR_PrintRect` procedure

GLIR: GLIR_PrintRect

GLIR_PrintRect:

```
Prints a rectangle on the terminal.

Arguments:

        a0: Row of the top left corner

        a1: Col of the top left corner

        a2: Signed height of the rectangle

        a3: Signed width of the rectangle

        a4: Colour to print with

        a5: Address of the null-terminated string to print with; if 0 uses the unicode full block

            char (█) as default

Returns:

        None
```

Pathfinder Visualizer General Flow

1. Build the map
2. Draw the map on the terminal
3. Run A* search from the start cell
4. If a solution path is found, draw the solution path in purple
5. Redraw the start and goal cells

Demonstration

Build the map

Map:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Map Buffer:

Value	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Initialization

Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

parent g h

Closed List

-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree

Open List - array

In this lab, A* must visit adjacent cells in the following order: left, right, top, and bottom

Visit the Start Cell

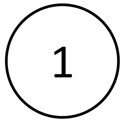
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

-1	0	0	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	**1**
Index	0

Expand cell 1

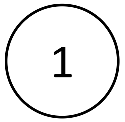
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

-1	0	0	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	**1**
Index	0

Step 1 - remove cell 1 from open list

No top adjacent cell

Right adjacent cell is a water cell

Visit cell 0

Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree

0

Open List - array

Value	**0**
Index	0

Step 2 - visit left adjacent cell

Visit cell 6

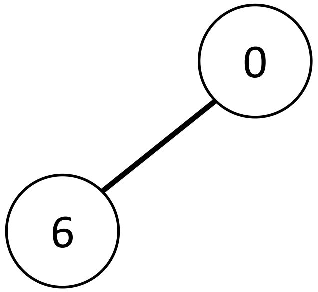
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	0	6
Index	0	1

Step 3 - visit bottom adjacent cell

Calculate f

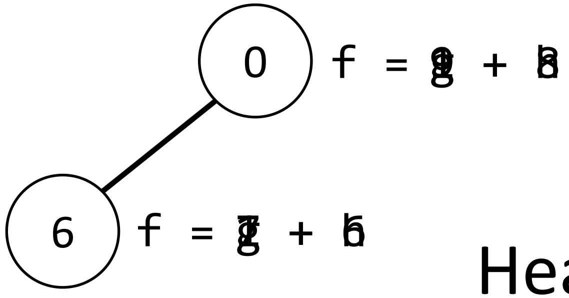
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

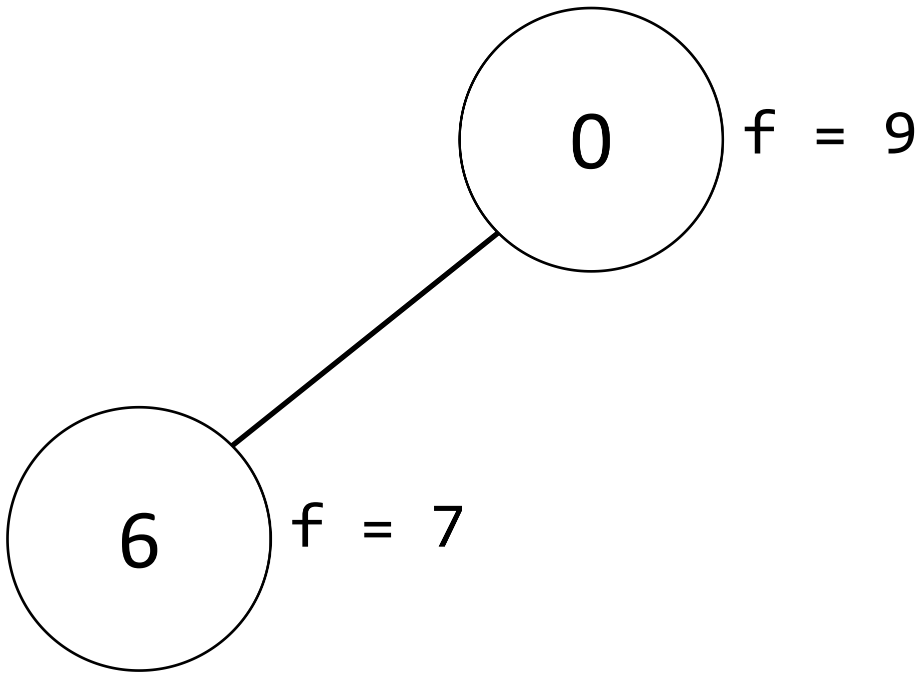
Value	0	6
Index	0	1

Heap property not satisfied

Heapify

Re-arrange elements in the open list such that it satisfies the heap property again

Open List - tree



Open List - array

Value	0	6
Index	0	1

Expand cell 6

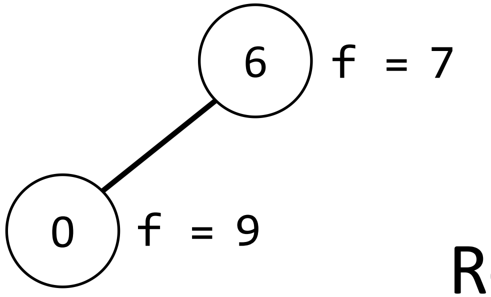
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
-1	0	0	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	**0**	**0**
Index	0	1

Remove cell 6 from open list

Visit cell 5

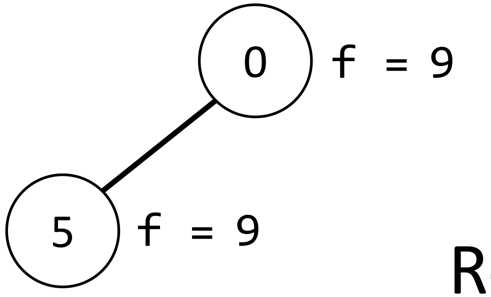
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
6	2	7	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	**0**	**5**
Index	0	1

Visit cell 1

Right adjacent cell is a water cell, skip

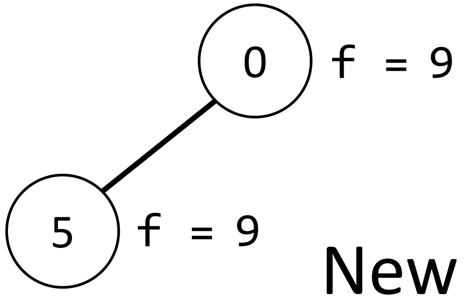
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

			6	2	7									
1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
6	2	7	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	0	5
Index	0	1

New g (2) greater than old g (0), skip

Visit cell 11

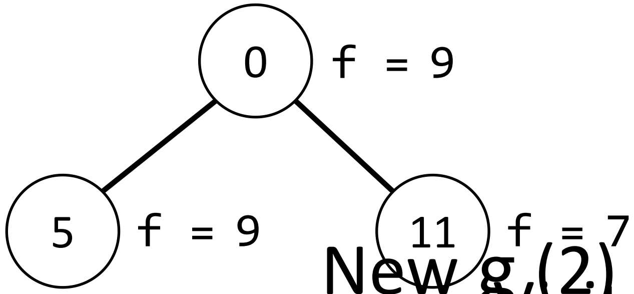
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
6	2	7	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	6	2	5	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	0	5	11
Index	0	1	2

New g (2) greater than old g (0), skip

Visit bottom adjacent cell

Visit cell 11

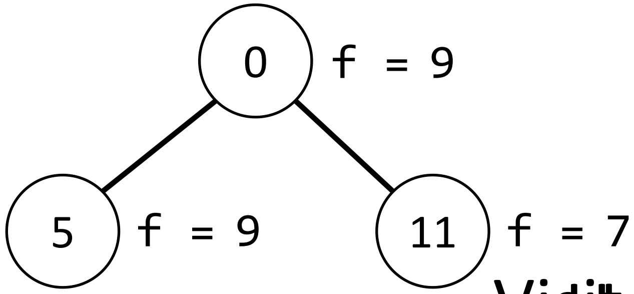
Map

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Closed List

1	1	8	1	0	7	-1	0	0	-1	0	0	-1	0	0
6	2	7	1	1	6	-1	0	0	-1	0	0	-1	0	0
-1	0	0	6	2	5	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0
-1	0	0	-1	0	0	-1	0	0	-1	0	0	-1	0	0

Open List - tree



Open List - array

Value	**0**	**5**	**11**
Index	0	1	2

Visit bottom adjacent cell

Heapify the open list

Exercise

- Try tracing the A* pseudocodes with the previous example