Graph Search

ROB 102: Introduction to AI & Programming

Lab Session 6

2021/11/12

Administrative

Project 3 is due Monday, November 22nd, at 11:59 PM!

Demo day is the same Monday (Nov 22).

Make the code updates described here:

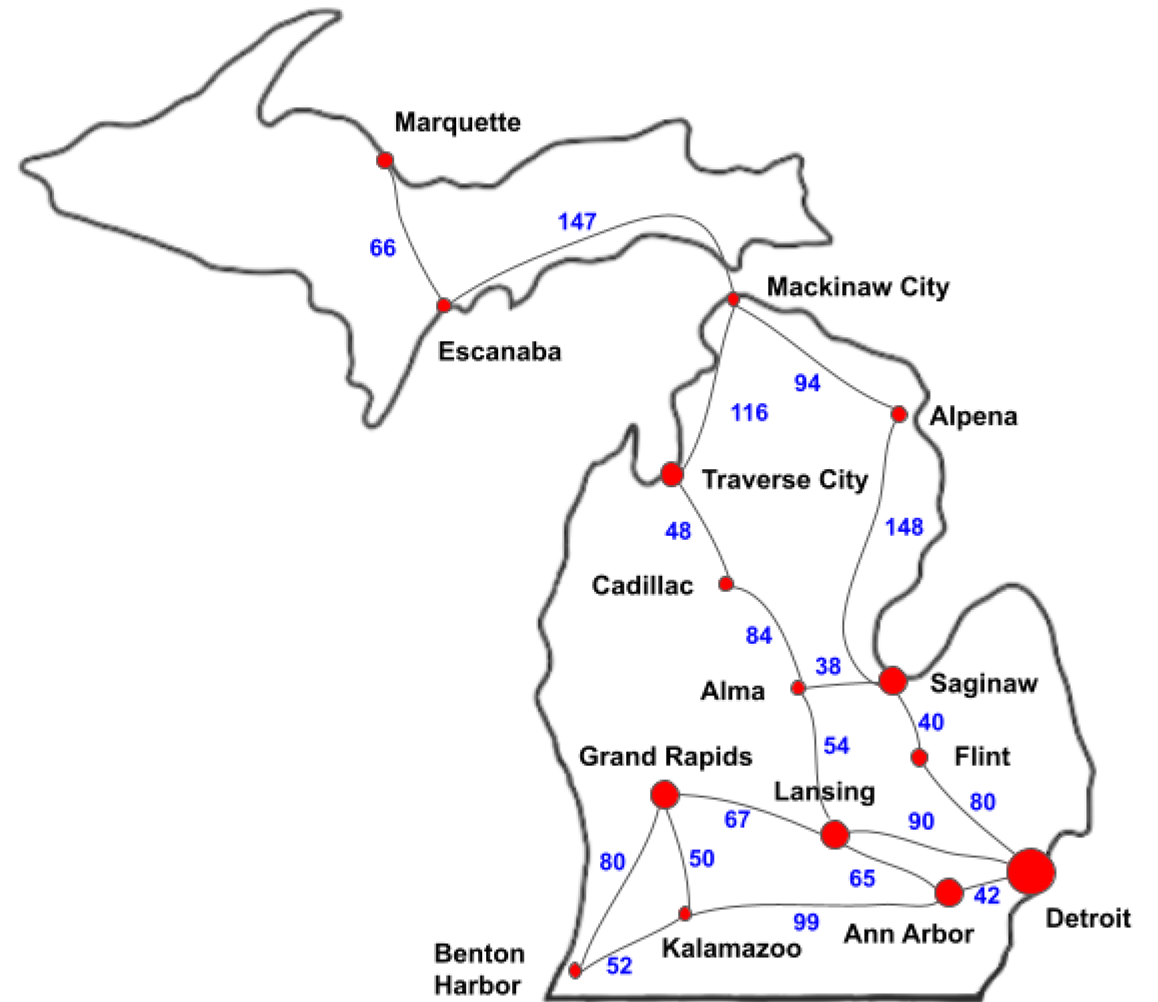
https://robotics102.github.io/projects/a3.html#code_changes

Today

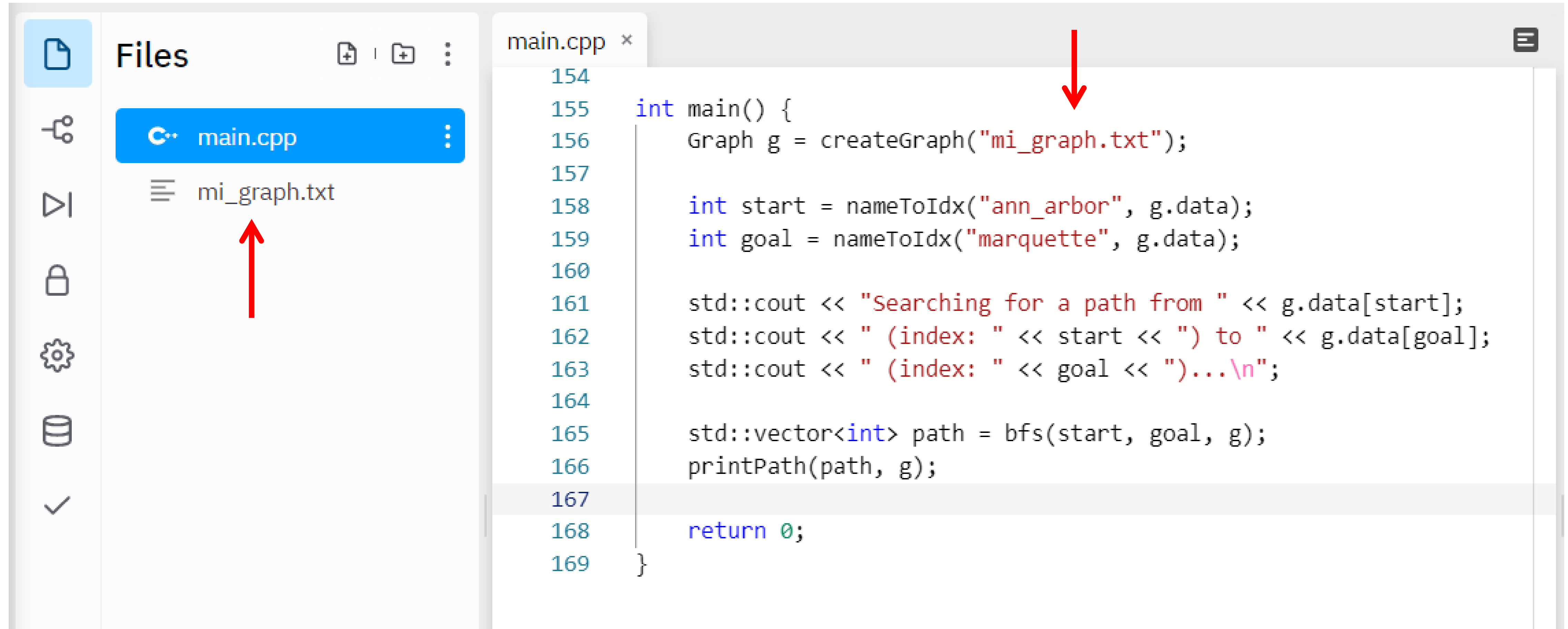
1. Breadth First Search (BFS) in-class activity code
2. Graph search code for Project 3
3. Running path planning on the robot
4. Time to work on in-class activity / Project 3.

Breadth First Search

Write C++ code to perform BFS. Your algorithm should find a path between any start and end node.



BFS in C++



BFS in C++

Things to consider:

- How can we represent and store the node data?
- How do we create a visit list?

BFS in C++

You can modify the Graph struct to store the information we need to keep track of.

```
/**
 * TODO: Define any structs you might need here.
 */

struct Graph
{
    std::vector<std::string> data;
    std::vector<std::vector<int> > edges;
    std::vector<std::vector<float> > edge_costs;

    // TODO: Add any members you need to the graph.
};
```

In Project 3, you will modify `GridGraph` to store this data.

BFS in C++

Data is stored by index.

```
Graph g = createGraph("mi_graph.txt");

std::cout << "Graph data:\n";
for (int i = 0; i < g.data.size(); i++)
{
    std::cout << "\tindex " << i << ": " << g.data[i] << "\n";
}
```

In Project 3, indices where node data is stored should map to the corresponding cell the same way as in Project 2.

```
/**
 * TODO: Define any structs you might need here.
 */

struct Graph
{
    std::vector<std::string> data;
    std::vector<std::vector<int> > edges;
    std::vector<std::vector<float> > edge_costs;

    // TODO: Add any members you need to the graph.
};
```

Console Shell

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Graph data:
    index 0: alma
    index 1: alpena
    index 2: ann_arbor
    index 3: benton_harbor
    index 4: cadillac
    index 5: detroit
    index 6: escanaba
    index 7: flint
    index 8: grand_rapids
    index 9: kalamazoo
    index 10: lansing
    index 11: mackinaw_city
    index 12: marquette
    index 13: saginaw
    index 14: traverse_city
> 
```

BFS in C++

To get a vector containing the indices of the neighbors of node `i`:

```
getNeighbors(i, g)
```

To get the distances between `i` and each neighbor:

```
getEdgeCosts(i, g)
```

```cpp
int idx = nameToIdx("alma", g.data);
std::vector<int> nbrs = getNeighbors(idx, g);
std::vector<float> dists = getEdgeCosts(idx, g);

std::cout << "Neighbors of " << g.data[idx] << " (index: " << idx << ")\n";
for (int n = 0; n < nbrs.size(); n++)
{
    int nbr = nbrs[n];
    std::cout << "\t" << g.data[nbr] << " (index: " << nbr << ") ";
    std::cout << "Distance: " << dists[n] << "\n";
}
```

Console Shell

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Neighbors of alma (index: 0)
    lansing (index: 10) Distance: 54
    saginaw (index: 13) Distance: 38
    cadillac (index: 4) Distance: 84
>
```

BFS in C++

```
int getParent(int idx, Graph& g)
{
    // TODO: This function should return the index of the parent of
    the node at idx.
    // If the node has no parent, return -1.
    return -1;
}

void initGraph(Graph& g)
{
    // TODO: Initialize any data you need for graph search.
}

std::vector<int> bfs(int start, int goal, Graph& g)
{
    initGraph(g);
    std::vector<int> path;  // Put your final path here.

    std::queue<int> visit_list;

    // TODO: Perform Breadth First Search over the graph g.

    return path;
}
```

Return the index of the parent of the node at index `idx`.

Initialize the new data structures you added to Graph to prepare for graph search.

Implement your algorithm for BFS here.

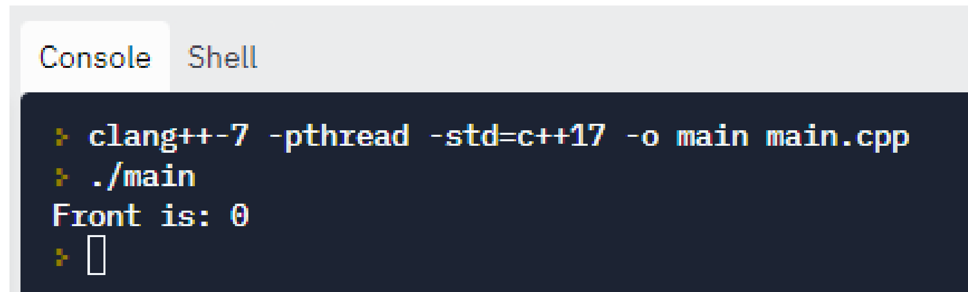
BFS in C++: Queues

In BFS, we can use a queue for our visit list.

With a queue, we add new elements to the end of the list, and pop elements off the front of the list (first in, first out)

```
std::queue<int> q;
q.push(0);
q.push(2);
q.push(4);
q.push(6);

std::cout << "Front is: " << q.front() << "\n";
```



BFS in C++: Queues

In BFS, we can use a queue for our visit list.

With a queue, we add new elements to the end of the list, and pop elements off the front of the list (first in, first out)

```
std::queue<int> q;
q.push(0);
q.push(2);
q.push(4);
q.push(6);

std::cout << "Front is: " << q.front() << " Queue size: " << q.size() << "\n";

while (!q.empty())
{
    std::cout << "Front: " << q.front();
    q.pop();
    std::cout << " Queue size: " << q.size() << "\n";
}
```

Console Shell

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Front is: 0 Queue size: 4
Front: 0 Queue size: 3
Front: 2 Queue size: 2
Front: 4 Queue size: 1
Front: 6 Queue size: 0
>
```

Project 3: Graph Search

```
struct GridGraph
{
    GridGraph() :
        width(-1),
        height(-1),
        origin_x(0),
        origin_y(0),
        meters_per_cell(0),
        collision_radius(0.15),
        threshold(-100)
    {
    };

    int width, height;                      // Width and height of the map in cells.
    float origin_x, origin_y;               // The (x, y) coordinate corresponding to cell (0, 0) in meters.
    float meters_per_cell;                  // Width of a cell in meters.
    float collision_radius;                 // The radius to use to check collisions.
    int8_t threshold;                       // Threshold to check if a cell is occupied or not.

    std::vector<int8_t> cell_odds;          // The odds that a cell is occupied.
    std::vector<float> obstacle_distances;  // The distance from each cell to the nearest obstacle.

    /**
     * TODO (P3): Define a vector named nodes which stores all the nodes in
     * your graph. The nodes should be of the type you defined above.
     */
};
```

← Modify `GridGraph` to store the node information you need

`include/autonomous_navigation/utils/graph_utils.h`

Project 3: Graph Search

We provide a function to find the neighbors of the cell at a given index.

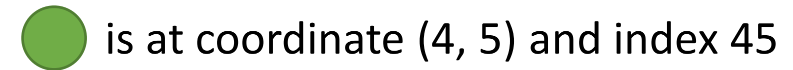
```
/**
 * Finds the neighbors of the cell at the given index.
 * @param  idx    The index of the cell in the graph data.
 * @param  graph  The graph the cell belongs to.
 * @return  A vector containing the indices of each of the valid neighbors.
 */
std::vector<int> findNeighbors(int idx, const GridGraph& graph);
```

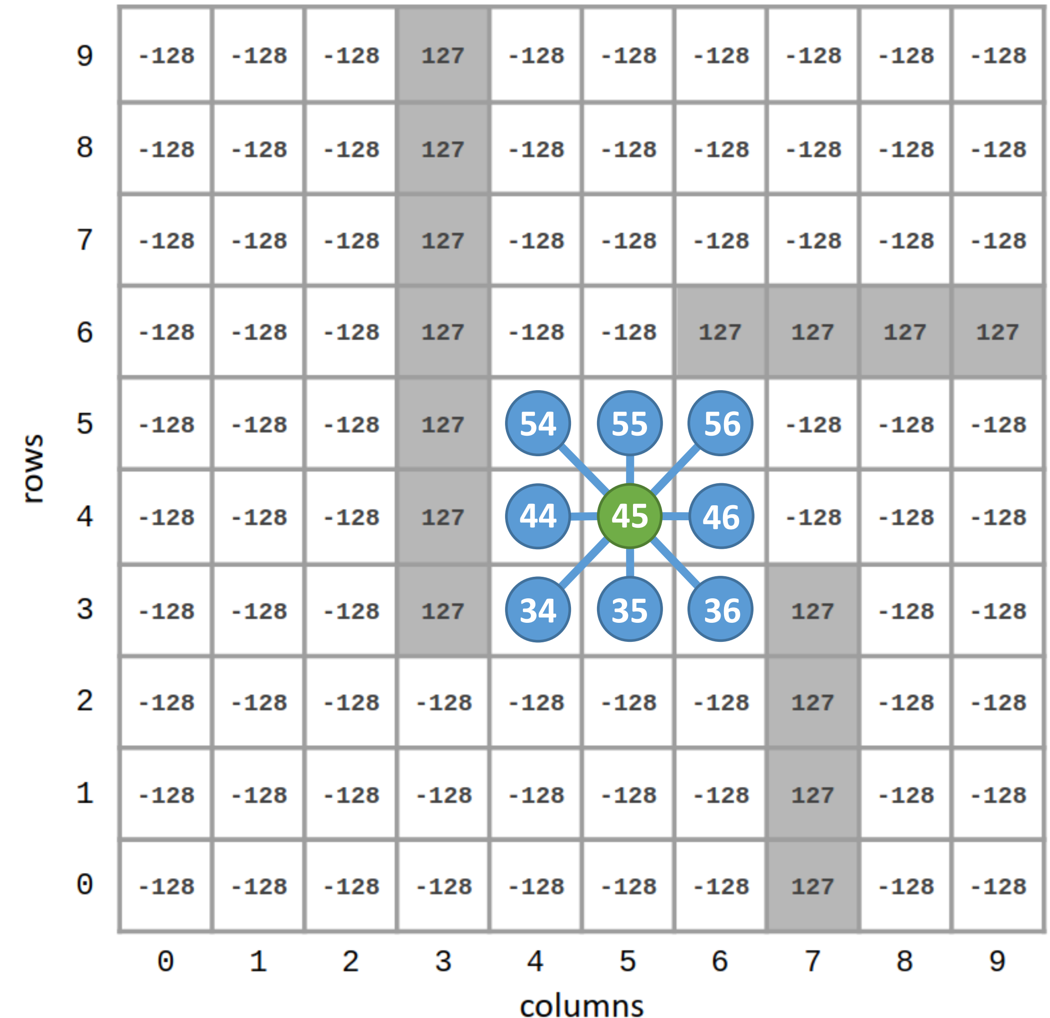
To find its neighbors:

```
nbrs = findNeighbors(45, graph)
```

`nbrs` contains the index of the 8 neighbor cells:

```
[54, 55, 56, 46, 36, 35, 34, 44]
```





You must calculate the distance to the neighbors!

Project 3: Graph Search

We provide a function to find the neighbors of the cell at a given index.

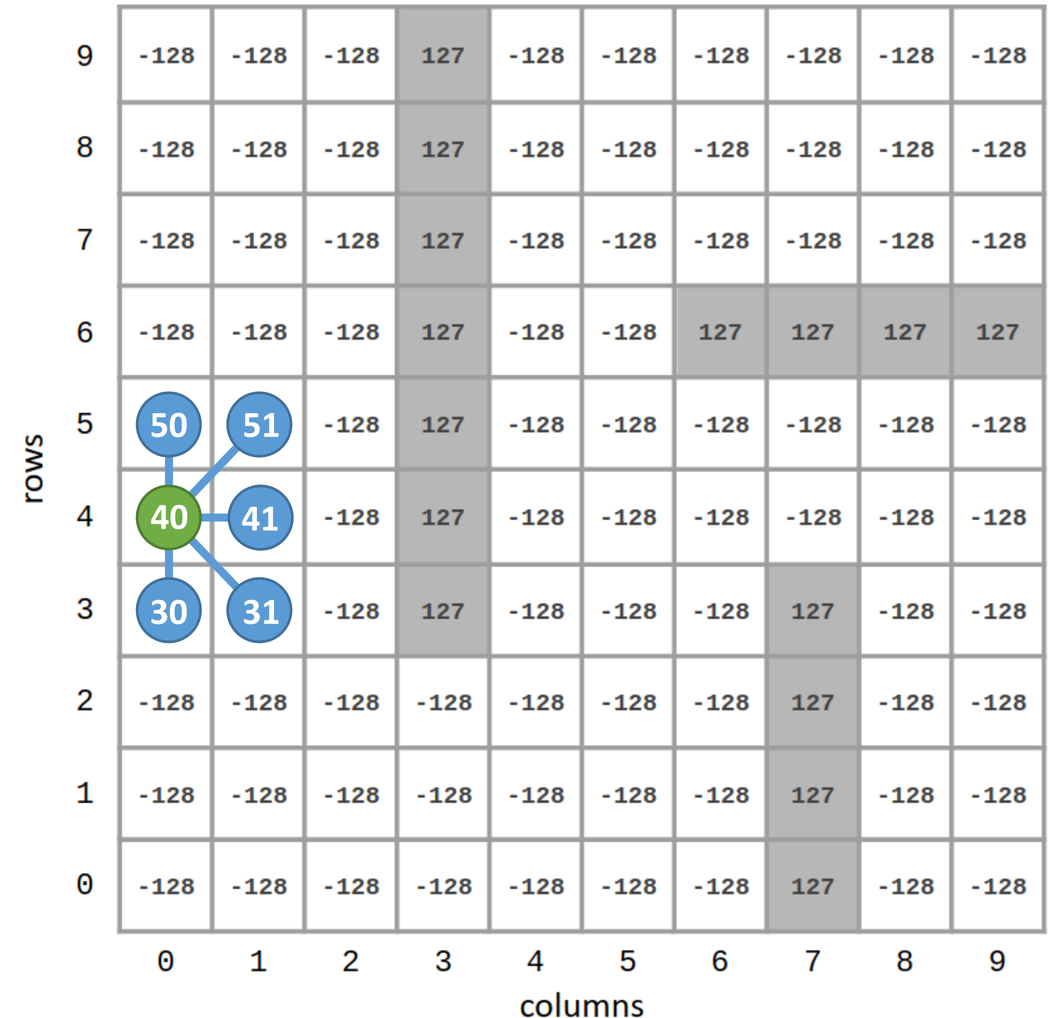
```
/**
 * Finds the neighbors of the cell at the given index.
 * @param  idx    The index of the cell in the graph data.
 * @param  graph  The graph the cell belongs to.
 * @return  A vector containing the indices of each of the valid neighbors.
 */
std::vector<int> findNeighbors(int idx, const GridGraph& graph);
```

Edge cells have less than 8 neighbors:

`nbrs = findNeighbors(40, graph)`

`nbrs` contains the index of the 5 neighbor cells:

`[50, 51, 41, 31, 30]`



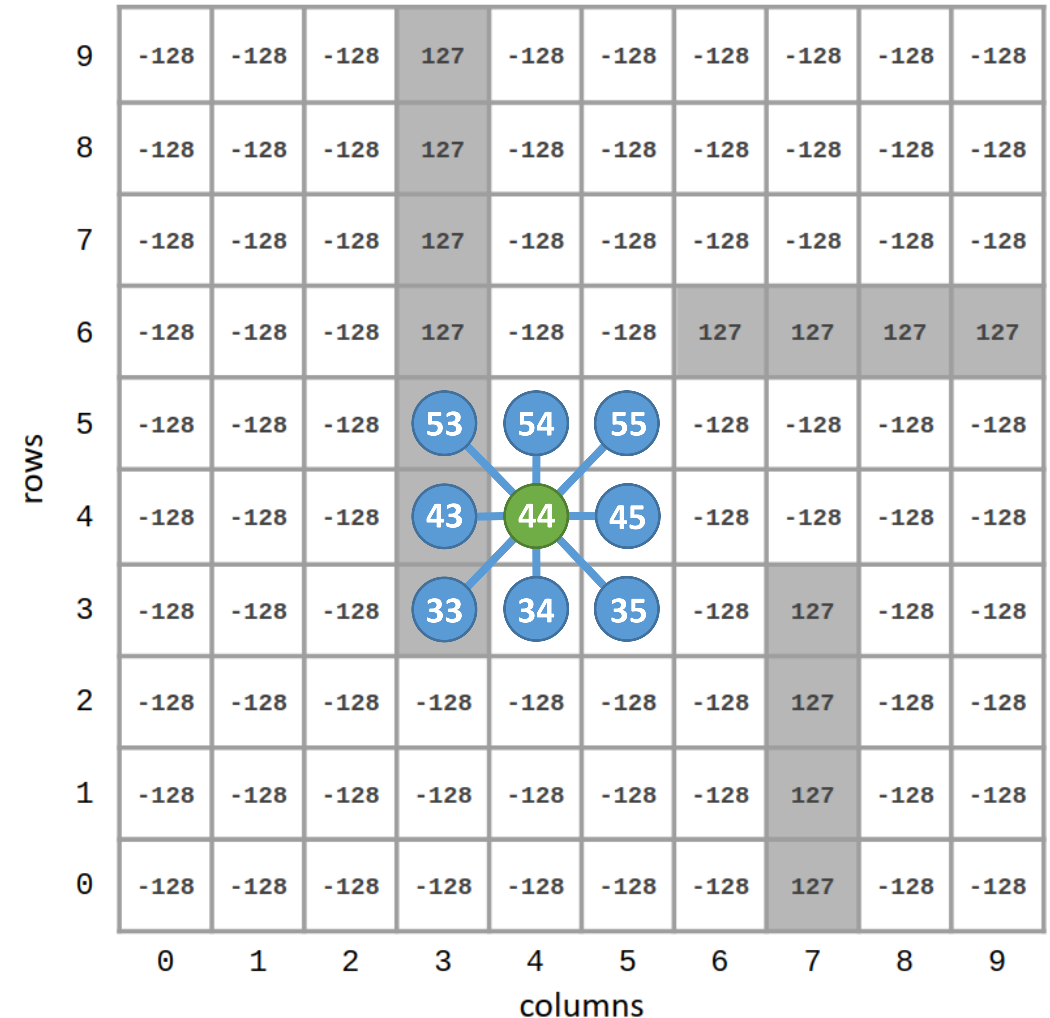
Project 3: Graph Search

`findNeighbors()` does not check if the neighbors are occupied or in collision!

```
/**
 * Checks whether the provided index in the graph is within the defined
 * collision radius of an obstacle using the distance transform.
 *
 * Warning: Distance transform values must be stored in graph.obstacle_distances
 * for this function to work.
 * @param  idx    The index of the cell in the graph data.
 * @param  graph  The graph the cell belongs to.
 */
bool checkCollisionFast(int idx, const GridGraph& graph);
```

Use `checkCollisionFast(idx, graph)` to check for collisions using the distance transform.

Cells which would result in a collision with an obstacle should not be added to the visit list.



Project 3: Graph Search

```
std::vector<Cell> breadthFirstSearch(GridGraph& graph, const Cell& start, const Cell& goal,
                                     std::function<void(int, int)> showVisitedCell)
{
    std::vector<Cell> path;  // The final path should be placed here.

    initGraph(graph);  // Make sure all the node values are reset.

    int start_idx = cellToIdx(start.i, start.j, graph);

    /**
     * TODO (P3): Implement BFS.
     */

    return path;
}
```

← Write your code here!

src/graph_search/graph_search.cpp

Path Planning on the Robot

To compile the code on the robot, do:

```
cd build/
cmake -DOMNIBOT=On ..
make
```

Important! Robot code is only compiled on the robot (not in Docker)

Start the localization (you might want to use NoMachine and the BotGUI):

```
~/botlab-bin/launch_botlab.sh -m [PATH/TO/MAP]
```

Use a fixed map.

Run your code on the robot:

Run motion control this time!

```
./robot_plan_path [PATH/TO/MAP] [goal_x goal_y]
```

Can provide goal x and y position in meters

Use the same map to compute the field

TODO Today:

1. Work on repl.it graph search activity (optional but encouraged)
2. Work on Project 3