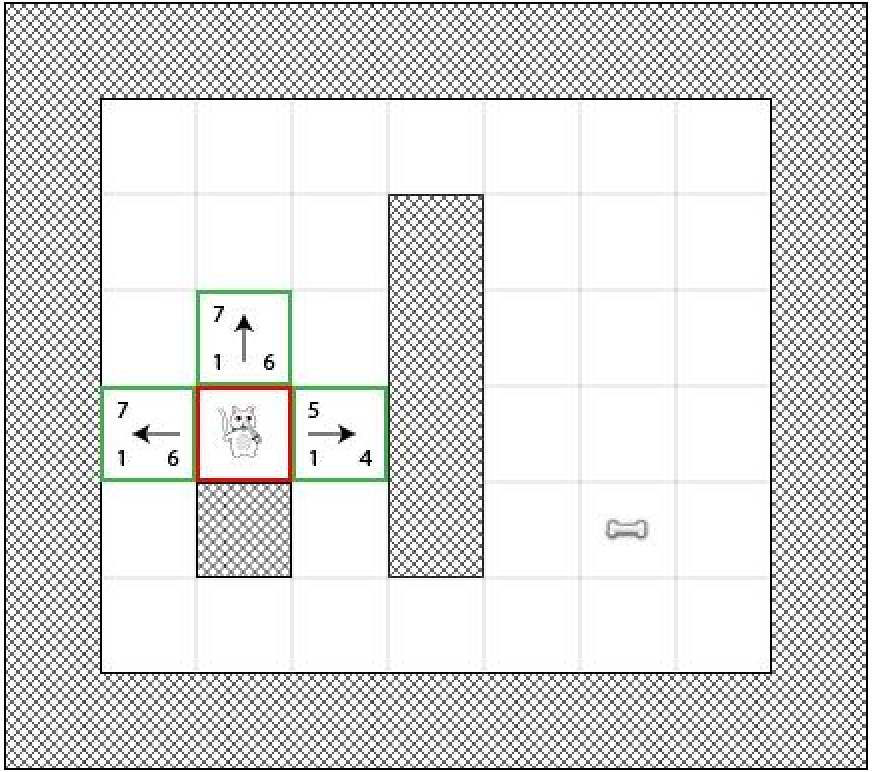
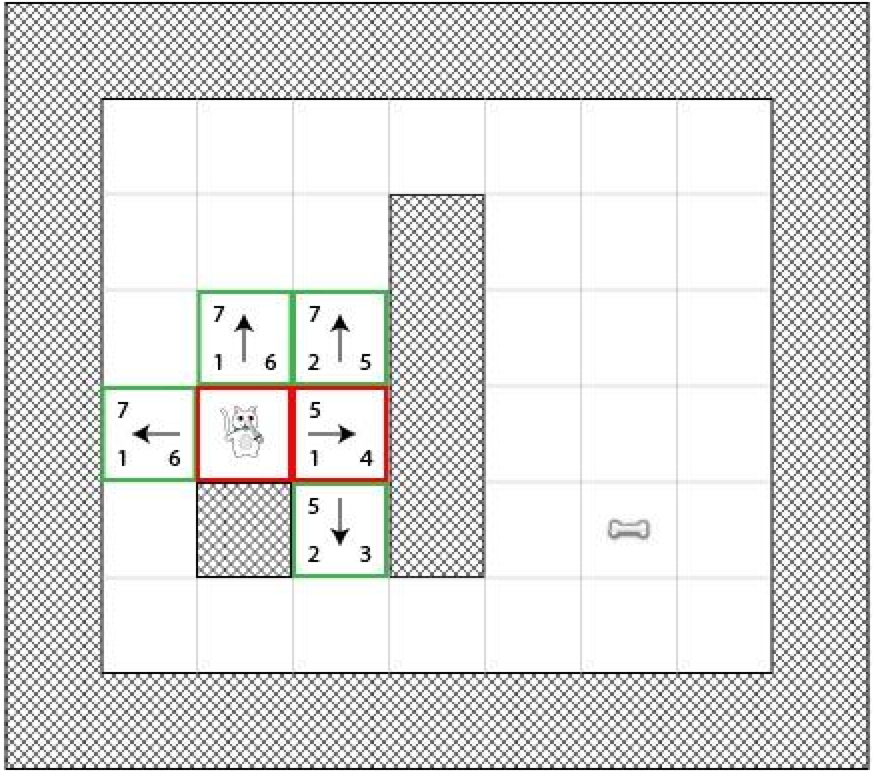
Game dev: A* Module

Ricard Pillosu - UPC

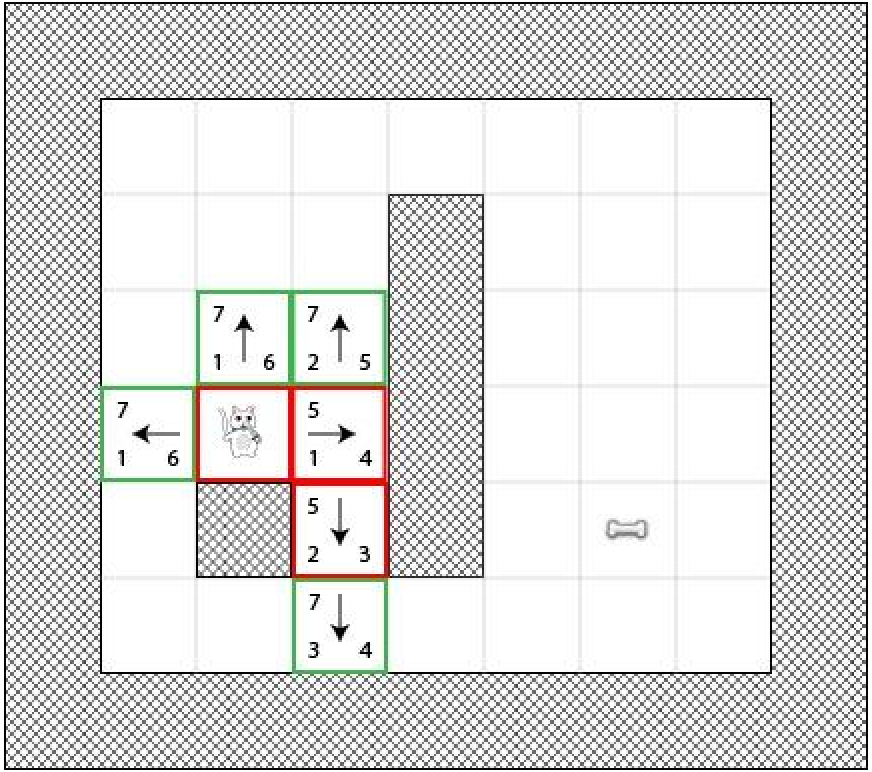
Formal A*: Syntax

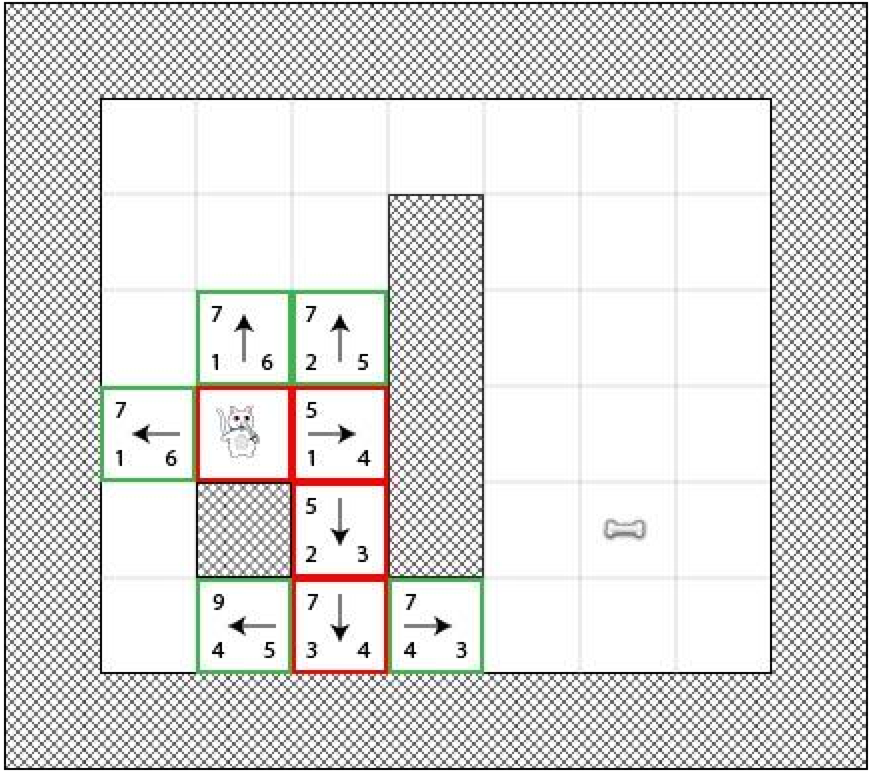
- We have a list of open (frontier) and closed (visited) nodes
- We'll give each square a score **F = G + H** where:
    - **G** is the movement cost from the start point A to the current square. This will increase as we get farther away from the start point.
    - **H** is the estimated movement cost from the current square to the destination point. This is often called the *heuristic* because we don't really know the cost yet – it's just an estimate.



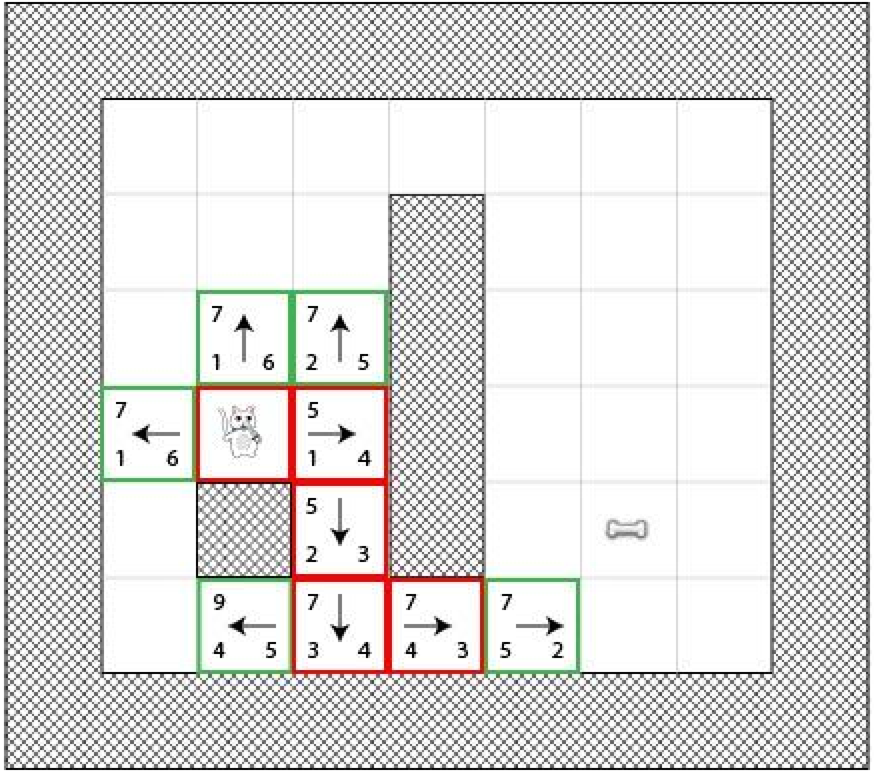


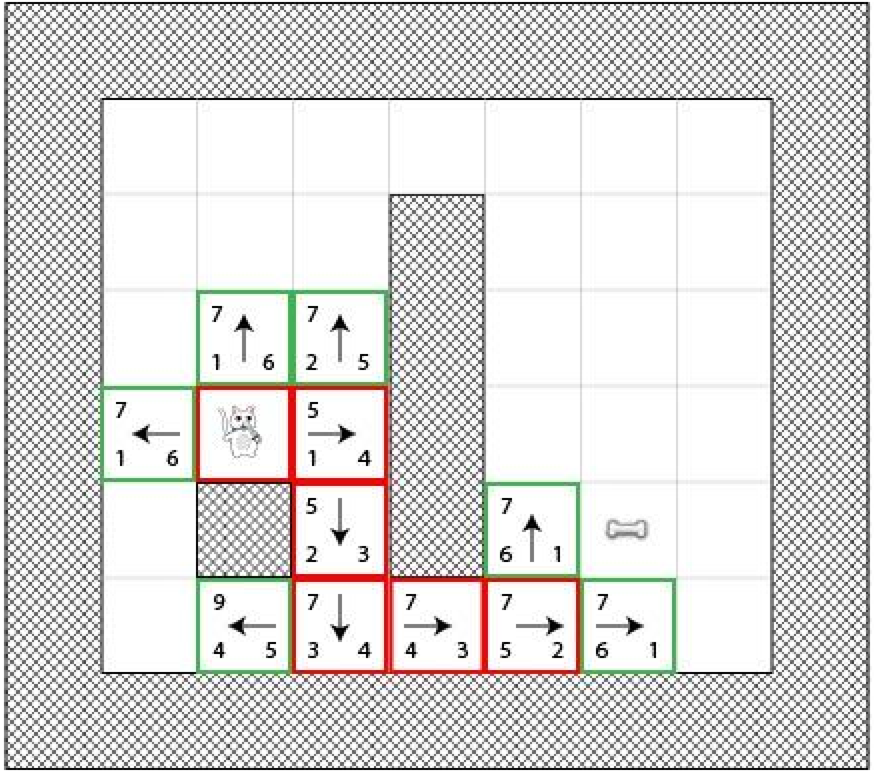
A* Progression



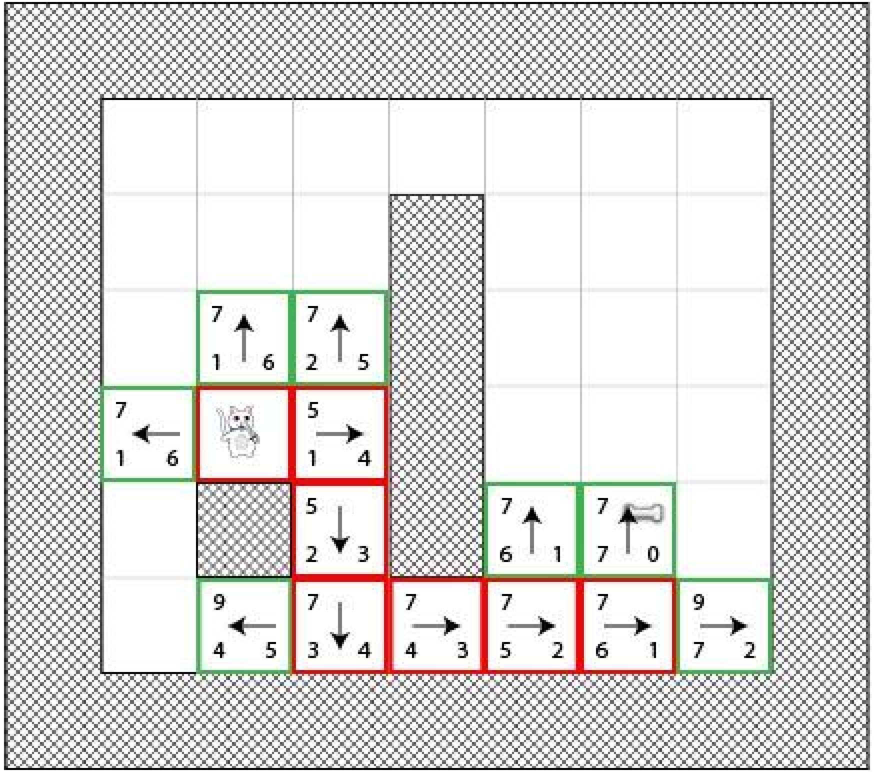


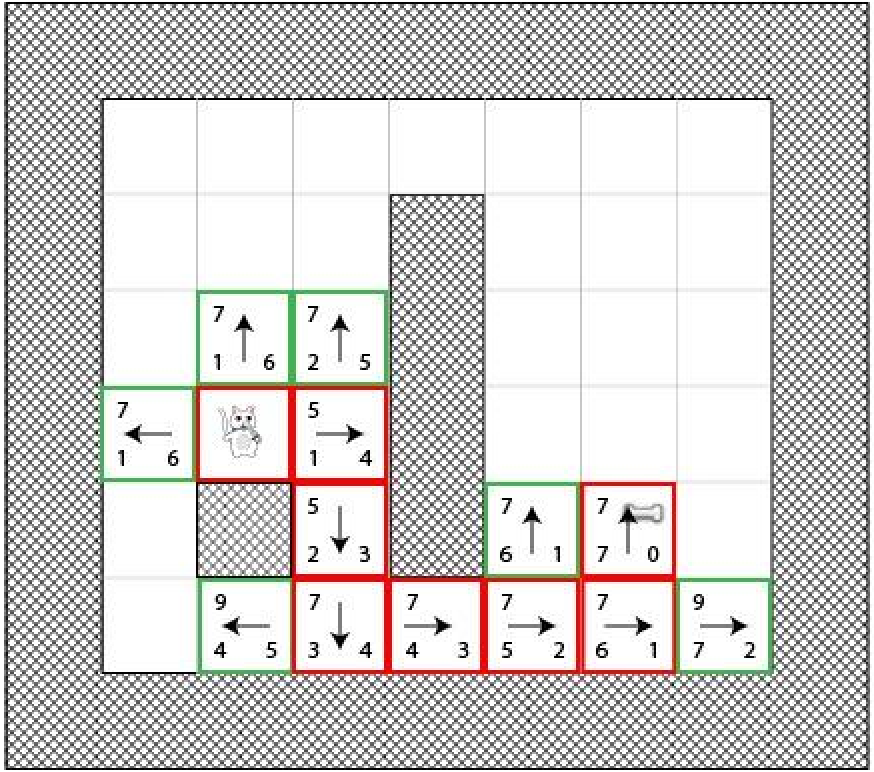
A* Progression





A* Progression





A* Progression

Implementing A*: Strategy

- Use a full module
- Create the path and store it in the module
- Will need supporting structures:
  - **PathNode**: Properties and methods about a single node
  - **PathList**: Handles a bunch of nodes

Implementing A*: The main module

Three main methods:

- **SetMap**: Received all the info about the tiles and its walkability
  - `void SetMap(uint width, uint height, uchar* data)`
- **CreatePath**: Request to have a path from A to B
  - `int CreatePath(const iPoint& origin, const iPoint& destination)`
- `GetLastPath`: Returns order path step by step
  - `const p2DynArray<iPoint>* GetLastPath() const`

Implementing A*: The main module

Three utility methods:

- **CheckBoundaries**: return true if pos is inside the map boundaries
  - `bool CheckBoundaries(const iPoint& pos) const`
- **IsWalkable**: returns true is the tile is walkable
  - `bool IsWalkable(const iPoint& pos) const`
- **GetTileAt**: return the walkability value of a tile
  - `uchar GetTileAt(const iPoint& pos) const`

Implementing A*: PathNode Structure

- It contains the **g, h, x, y** and **parent**
- Convenient constructors
- **FindWalkableAdjacents**: Fills a list of adjacent tiles that are walkable
  - `uint FindWalkableAdjacents(PathList& list_to_fill) const`
- **Score**: Basically returns g + h
  - `int Score() const`
- **CalculateF**: Recalculates F based on distance to destination
  - `int CalculateF(const iPoint& destination)`

Implementing A*: PathList Structure

- It contains a linked list of *PathNode* (not *PathNode**)
- **Find**: Returns the node item if a certain node is in this list already (or NULL)
  - `p2List_item<PathNode>* Find(const iPoint& point) const`
- **GetNodeLowestScore**: Returns the Pathnode with lowest score in this list or NULL if empty
  - `p2List_item<PathNode>* PathList::GetNodeLowestScore() const`

TODO 1

“if origin or destination are not walkable, return -1”

- To simplify we will reject paths that begin or end in not walkable tiles
- We return -1 in case of invalid request

TODO 2

“Create two lists: open, close. Add the origin tile to open. Iterate while we have a tile in the open list”

```
// ----------------------------------------------------------------------
// Helper struct to include a list of path nodes
// ----------------------------------------------------------------------
struct PathList
{
 // Looks for a node in this list and returns it's list node or NULL
 p2List_item<PathNode>* Find(const iPoint& point) const;
 // Returns the Pathnode with lowest score in this list or NULL if empty
 p2List_item<PathNode>* GetNodeLowestScore() const;
 // The list itself, note they are not pointers!
 p2List<PathNode> list;
};
```

TODO 3

"Move the lowest score cell from open list to the closed list"

- Moving means copying and then destroying the old one
- To remove from a list use the *Del()* methods of the list

TODO 4

“If we just added the destination, we are done! Backtrack to create the final path. Use the Pathnode::parent and Flip() the path when you are finish”

- Basically write the exit of that infinite loop
- When we find the destination, we go tracking down tiles using the Parent.
- Backtracking means that the path is from destination -> origin.
- Just Flip() it :)

TODO 5

"Fill a list of all adjacent nodes"

- Simple enough

TODO 6

"Ignore nodes in the closed list. If it is NOT found, calculate its F and add it to the open list. If it is already in the open list, check if it is a better path (compare G). If it is a better path, Update the parent"

- This is the core of the algorithm!
- You could use "continue" C keyword for the first test.
- Now two choices: is this tile already in the open list ?
  - **True**: This might be a better path, compare G
  - **False**: Calculate the F and add it to the open list

Documentation

- Read carefully, G, H, F well explained here: http://www.raywenderlich.com/4946/introduction-to-a-pathfinding

Homework

- Implement movement in ***diagonal***
- Experiment with different ways to calculate **H** (see solutions)