Game ‘AI’ Part 2 Pathfinding

Admin

- Quiz
- Class Evals
- Project

Path Planning

- when you have “bad guys”
  - need to find your players.
-
- find a path from here to them
  - how?

Path Planning

- when you have “bad guys”
  - need to find your players.
-
- find a path from here to them
  - how?
  - particularly for tiled 2d games?

Path Planning

- when you have “bad guys”
  - need to find your players.
-
- find a path from here to them
  - how?
  - particularly for tiled 2d games?
    - graph search!!

Brute Force Techniques

- Some games might use brute force
  - depth first
  - breadth first
- Some might use heuristic algorithm like A*

Setting up the graph

- Very important to setup graph well in your game
  - best search in the world won't help AI chars who have lousy graphs.
  - trade offs and design decisions

Common Setup: Visibility Graph

- VGraph common for games as well as robotics

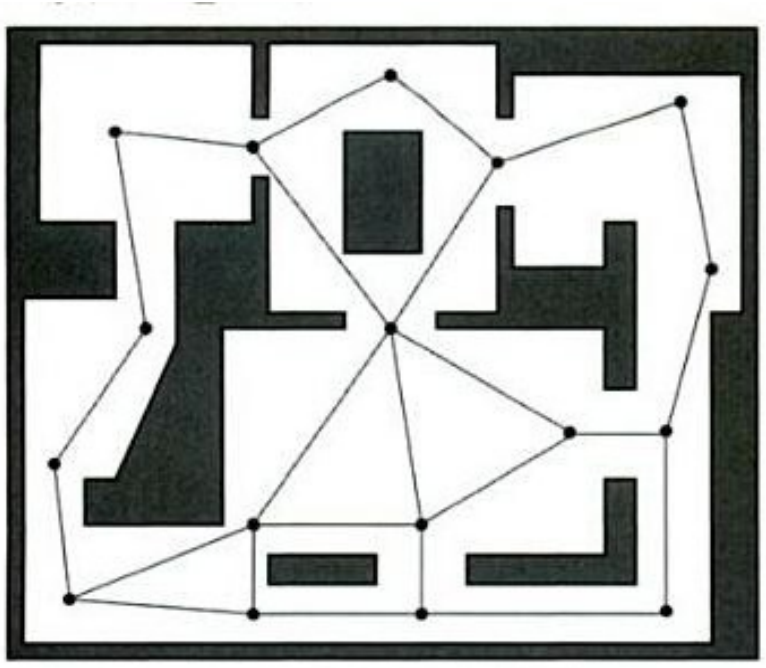


Figure 8.1. Points of visibility navigation graph

Alternative: Expand Geometry

- Expanded/dilated geometry
  - more useful for geometry than tile based games
  - dilate by at least size of NPC
  - build vgraph from result

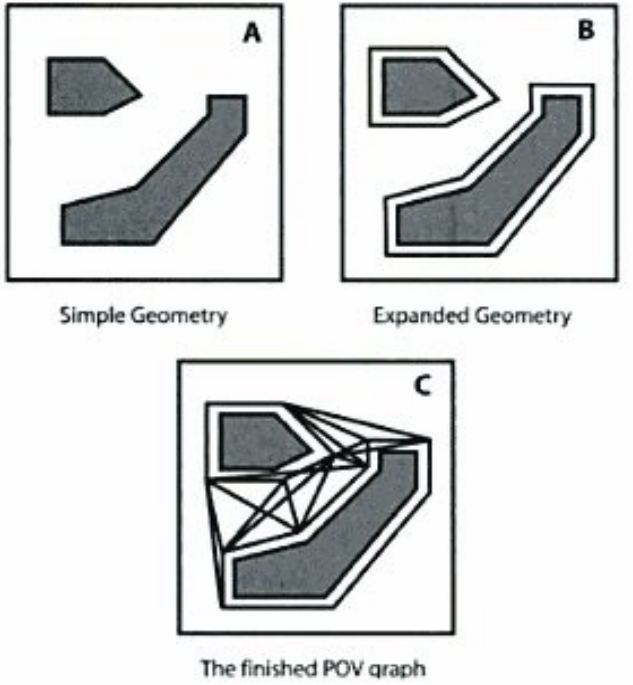
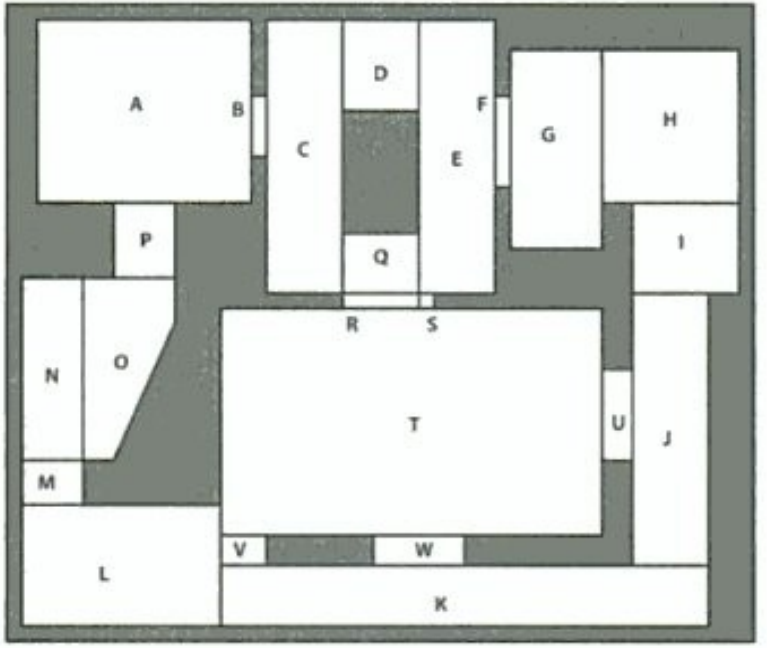


Figure 8.2. Creating a POV using expanded geometry

NavMesh

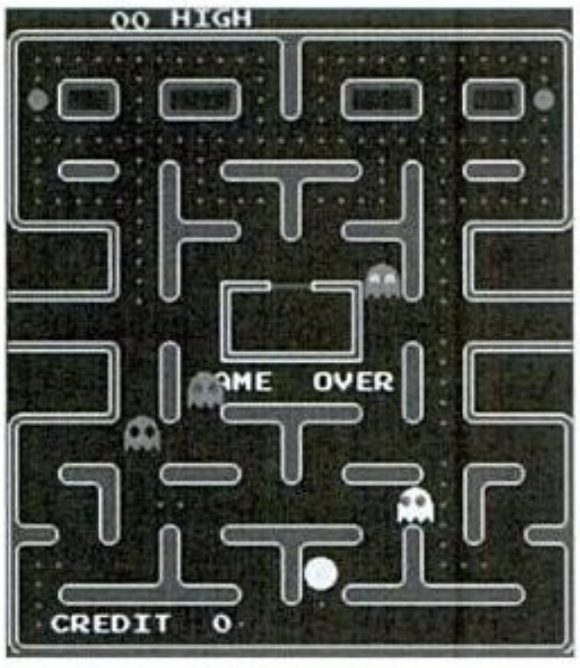
- Nav Mesh approach
  - good for 3d
  - npcs can wander freely within area
    - travel from area to area along prescribed boundaries
    -
  - efficient



Course Granulation for graphs

- Coarsely Granulated graphs
  - only a few nodes
  - setup by hand
  - advantages
    - very space efficient
    - very easy to search
  - disadvantages
    - potential for blind spots (p 338)
    - hard to use in free movement games
    - leads to unsightly zigzag paths
    - developer time required.
  - What sorts of game(s) is ideal for this sort of graph?

Course grained graph poster child



Finely Grained Graphs

- Finely grained graphs
  alternative to course grained
    - pict from page 339

Finely Grained Graphs

- Finely grained graphs
  - advantages
    - much smoother motion
    - no blind spots
    - ideal for tile based layouts.
  - disadvantages
    - larger memory
    - larger search times
      - Though with modern machines this is less of an issue
      - Still ***lots*** faster than LLM style AI

Creating finely Grained graphs

- use flood fill algorithm
  - start with seed point
  - expand node/edges outward in all available directions
  - continue from graph edges
  - till navigable space full

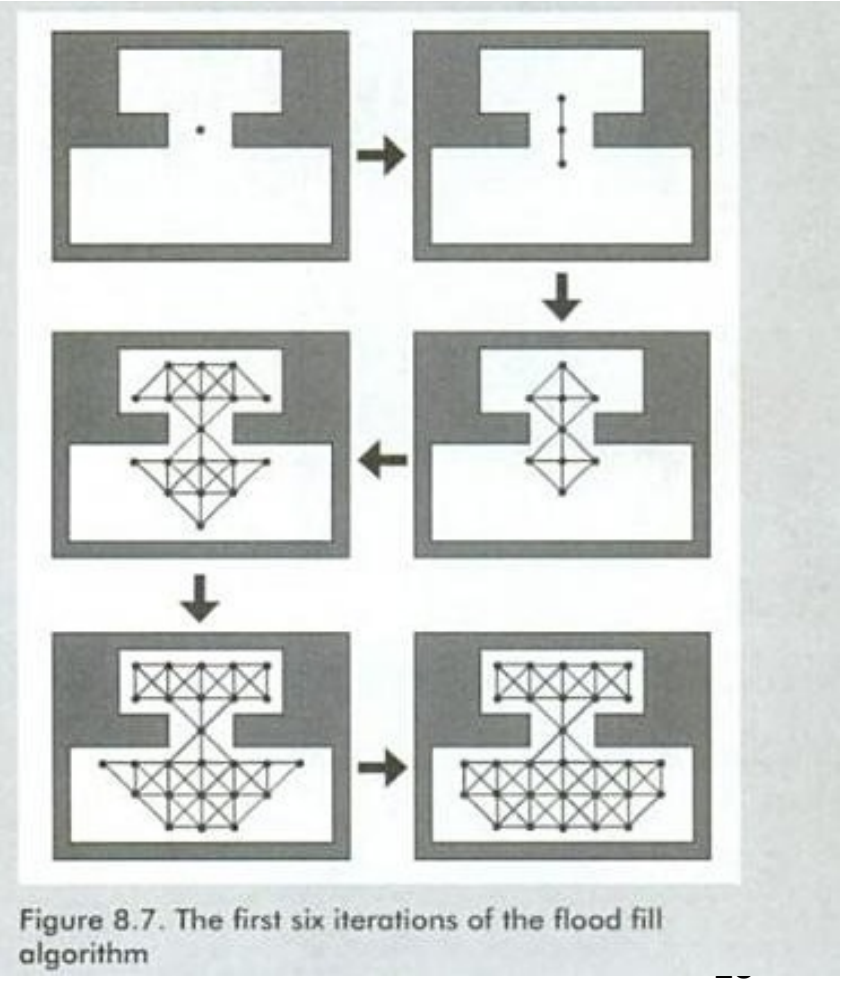


Figure 8.7. The first six iterations of the flood fill algorithm

Lets try it

- First we need to represent the map as a 2d array
  - For walls and floors we just need to mark the impassable layer in the array and leave the rest as default
  - How will we make a ‘2d array’ in ebitengine?

Lets try it

- First we need to represent the map as a 2d array
  - For walls and floors we just need to mark the impassable layer in the array and leave the rest as default
  - How will we make a ‘2d array’ in ebitengine?
  - maybe a slice of slices,
    - But go-tiled just unrolled it into a single long slice

Now Flood Fill

- Once you have the map as an array
- Lets look at zombies and cats.
  - http://inventwithpython.com/blog/2011/08/11/recursion-explained-with-the-flood-fill-algorithm-and-zombies-and-cats/
- Look at their super simple flood fill
  - We need to make sure we don’t go off the edge of the map.
  - But otherwise our solution will be similar
- Once you have the graph, you are ready to graph search.

Items and Graphs

- not just places, but items on graphs
  - items players can pick up
  - or AI npcs as well.
  - make your AI look smarter
    - put items on paths AI chars travel most
      - player travel or not.

How can we build our graph?

- We need to build a graph with flood fill, then do a graph search to find a path.
- When I did this with python students had to implement recursive flood fill.
- And of course all of you completed comp250 so you know how to implement a depth-first or breath first search
  - Maybe not A*
- If you had your 'druthers' how would you do it?

How can we build our graph?

- We need to build a graph with flood fill, then do a graph search to find a path.
- When I did this with python students had to implement recursive flood fill.
- And of course all of you completed comp250 so you know how to implement a depth-first or breath first search
  - Maybe not A*
- If you had your 'druthers' how would you do it?
  - 'Programmers are lazy' – meaning?

Someone else did it AKA don’t reinvent the wheel

- 'Programmers are lazy'
  - AKA don't reinvent the wheel
- The Awesome Ebitengine list on github
  - https://github.com/sedyh/awesome-ebitengine
- There are two promising pathfinding libraries.
  - That also build the graph.
  - https://github.com/SolarLune/paths - oh look resolv creator
  - https://github.com/quasilyte/pathing

SolarLune/paths

- First thing to know
  - The readme on github is a lying liar.
  - Epitome of stale comments/documentation.
  - One of us should update it in a pull request
    - In our spare time.

Pathing Example

- Path example
  - https://github.com/jsantore/AI_PathingDemo1
- We start out with the tiled map demo like a couple of (a few?) weeks ago
  - Change a couple of names and add a few things.
- Planned final result:
  - Display map
  - Put a pile of coins
  - Click anywhere on map, goblin appears and moves to coin pile without going in water or through walls

Main

- Annotated main
-
```
func main() {
    gameMap := loadMapFromEmbedded(path.Join("assets", "MapForPaths.tmx"))
    pathMap := makeSearchMap(gameMap) //this is two slides from here.
    searchablePathMap := paths.NewGridFromStringArrays(pathMap, gameMap.TileWidth, gameMap.TileHeight)
    searchablePathMap.SetWalkable('2', false) //sets the water tiles as not passable
    searchablePathMap.SetWalkable('3', false) //sets the wall tiles as not passable
    coins := makeCoinPile() //these two are on next slide
    nonPlayer := makeNPC() //these two are on next slide
    ebiten.SetWindowSize(gameMap.TileWidth*gameMap.Width, gameMap.TileHeight*gameMap.Height)
    ebiten.SetWindowTitle("Maps Embedded")
    ebitenImageMap := makeEbitenImagesFromMap(*gameMap)
    oneLevelGame := PathMapDemo{
    Level:        gameMap,
    tileHash:     ebitenImageMap,
    pathFindingMap: pathMap,
    coins:        coins,
    npc:          nonPlayer,
    pathMap:      searchablePathMap,
    }
    err := ebiten.RunGame(&oneLevelGame)
    if err != nil {
    fmt.Println("Couldn't run game:", err)
    }
}
```

Creating coins and goblin

-
-
```
func makeNPC() NonPlayerChar {
	picture := LoadEmbeddedImage("", "goblin.png")
	character := NonPlayerChar{
		pict: picture,
		xloc: -100, //put the NPC off screen originally
		yloc: -100,
	}
	return character
}
```

-
```
func makeCoinPile() coinPile {
	picture := LoadEmbeddedImage("", "coins.png")
	money := coinPile{
		pict:   picture,
		row:    12,
		column: 10,
	}
	return money
}
```

A lot like making the entities earlier – anything look out of place?

Make the search map

- In this one we iterate through the map and build the slice of strings representation the path library wants

```
func makeSearchMap(tiledMap *tiled.Map) []string {
	mapAsStringSlice := make([]string, 0, tiledMap.Height) //each row will be its own string
	row := strings.Builder{}
	for position, tile := range tiledMap.Layers[0].Tiles {
		if position%tiledMap.Width == 0 && position > 0 { // we get the 2d array as an unrolled one-d array
			mapAsStringSlice = append(mapAsStringSlice, row.String())
			row = strings.Builder{}
		}
		row.WriteString(fmt.Sprintf("%d", tile.ID))
	}
	mapAsStringSlice = append(mapAsStringSlice, row.String())
	return mapAsStringSlice
} //questions??
```

- strings.Builder is the efficient way to build a string from parts

Draw

```
func (demo PathMapDemo) Draw(screen *ebiten.Image) {
        drawOptions := ebiten.DrawImageOptions{}
        //draw map
        <snipped for slide – same as tiled map version>
        //draw gold
        drawOptions.GeoM.Reset()
        drawOptions.GeoM.Translate(float64(demo.coins.column*demo.Level.TileWidth),
                                   float64(demo.coins.row*demo.Level.TileHeight))
        screen.DrawImage(demo.coins.pict, &drawOptions)
        //draw goblin
        drawOptions.GeoM.Reset()
        drawOptions.GeoM.Translate(demo.npc.xloc, demo.npc.yloc)
        screen.DrawImage(demo.npc.pict, &drawOptions)
}
```

- Fairly standard draw – first map, now two objects, the goblin and the gold, goblin starts off screen

Check mouse

- New function checkmouse will
  - move the goblin to a mouseclick
  - and then ask path to plot path from goblin to gold

```
func checkMouse(demo *PathMapDemo) {
    if inpututil.IsMouseButtonJustPressed(ebiten.MouseButtonLeft) {
        mouseX, mouseY := ebiten.CursorPosition()
        demo.npc.xloc = float64(mouseX) //move the goblin to mouse loc
        demo.npc.yloc = float64(mouseY) //upper left of goblin
        startRow := int(demo.npc.yloc) / demo.Level.TileHeight
        startCol := int(demo.npc.xloc) / demo.Level.TileWidth
        startCell := demo.pathMap.Get(startCol, startRow)//start path from goblin's tile
        endCell := demo.pathMap.Get(demo.coins.column, demo.coins.row)
        demo.path = demo.pathMap.GetPathFromCells(startCell, endCell, false, false)
    }
}
```

- The two false arguments I'm passing to params in making the path are diagonal movement, and diagonal wall gaps allowed

The update has quite a bit that is new

```
func (demo *PathMapDemo) Update() error {
    checkMouse(demo)
    if demo.path != nil {
    pathCell := demo.path.Current() //get the current tile/cell in the path
    if math.Abs(float64(pathCell.X*demo.Level.TileWidth)-(demo.npc.xloc)) <= 2 &&
    math.Abs(float64(pathCell.Y*demo.Level.TileHeight)-(demo.npc.yloc)) <= 2 { //if we are now on the tile we need to be on
    demo.path.Advance() //make the current tile the next one in the path
    }
    direction := 0.0 //find the X direction to move to make upper left of goblin closer to upper left of tile
    if pathCell.X*demo.Level.TileWidth > int(demo.npc.xloc) {
    direction = 1.0
    } else if pathCell.X*demo.Level.TileWidth < int(demo.npc.xloc) {
    direction = -1.0
    }
    Ydirection := 0.0 //find Y direction to move to make upper left of goblin closer to upper left of tile
    if pathCell.Y*demo.Level.TileHeight > int(demo.npc.yloc) {
    Ydirection = 1.0
    } else if pathCell.Y*demo.Level.TileHeight < int(demo.npc.yloc) {
    Ydirection = -1.0
    }
    demo.npc.xloc += direction * 2  //move toward upper left of current tile
    demo.npc.yloc += Ydirection * 2
    }
    return nil
}
```

Let’s Try it (again?)

- And see how it works.
- Note that the library will do A*, you have to assign a cost for tiles in MakeSearchMap
  - I didn't since only my sand tile is passable.
  -

Reading/Watching/Learning

- A few bits of reading/watching for those who want to know how the library is implemented.
  - A tutorial from gamedeveloper
  - https://www.gamedeveloper.com/programming/toward-more-realistic-pathfinding
  - An older book chapter
  - https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1063&context=itbj
  - A video (I'll admit, I've only watched part of it – I've done it too much to watch all 12 minutes.
  - https://www.youtube.com/watch?v=i0x5fj4PqP4