Pathfinding

How to implement an A* algorithm

Pathfinding

- Almost every game requires pathfinding
- Agents must be able to find their way around the game world
- Pathfinding is not a trivial problem
- The fastest and most efficient pathfinding techniques tend to consume a great deal of resources

Searching for a Path

- A path is a list of cells, points, or nodes that an agent must traverse
- A pathfinding algorithm finds a path
  - From a start position to a goal position
- The following pathfinding algorithms can be used on (examples use grids for simplicity, same as your application)
  - Grids
  - Waypoint graphs
  - Navigation meshes

KU LEUVEN

Criteria for Evaluating Pathfinding Algorithms

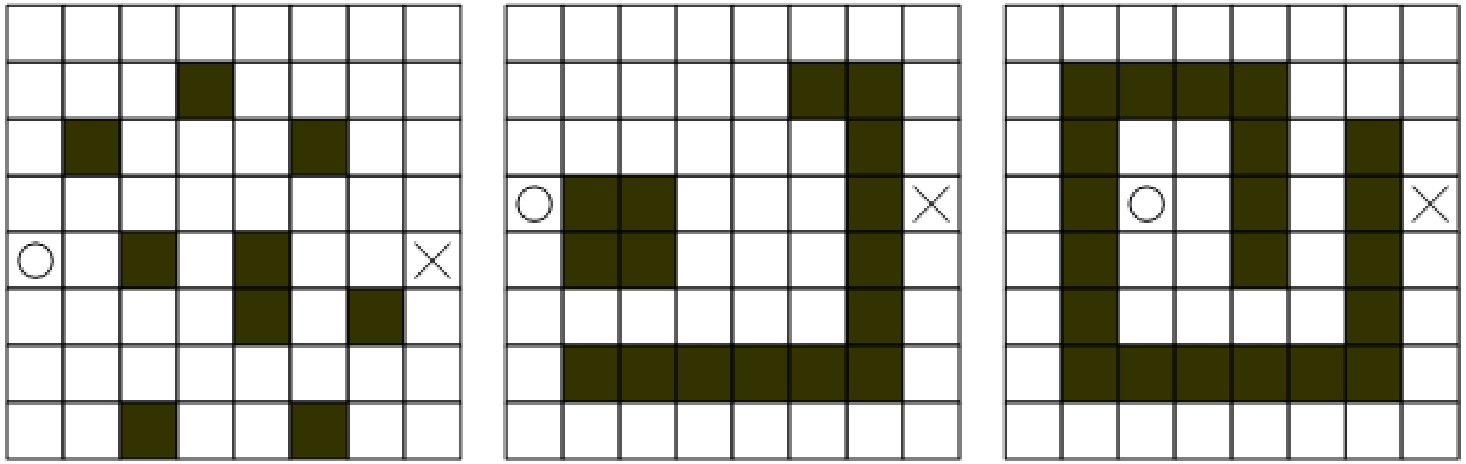
- Quality of final path
- Resource consumption during search
  - CPU and memory
- Whether it is a ***complete*** algorithm
  - A ***complete*** algorithm guarantees to find a path if one exists

Random Trace

- Simple algorithm
  - Agent moves towards goal
  - If goal reached, then done
  - If obstacle
    - Trace around the obstacle clockwise or counter-clockwise (pick randomly) until free path towards goal
  - Repeat procedure until goal reached

Random Trace (continued)

- How will Random Trace do on the following maps?



Random Trace Characteristics

- Not a ***complete*** algorithm
- Considers only 1 possible path
- Found paths are unlikely to be optimal
- Consumes very little memory

Understanding A*

- To understand A*
  - First understand Breadth-First, Best-First, and Dijkstra algorithms
- These algorithms use nodes to represent candidate paths

Understanding A*

```
class PlannerNode
{
public:
    PlannerNode    *m_pParent;
    int             m_cellX, m_cellY;
    ...
};
```

- The m_pParent member is used to chain nodes sequentially together to represent a path

Understanding A*

- All of the following algorithms use two lists
  - The ***open*** list
  - The ***closed*** list
- Open list keeps track of promising nodes
- When a node is examined from open list
  - Taken off open list and checked to see whether it has reached the goal
- If it has not reached the goal
  - Used to create additional nodes
  - Then placed on the closed list

Overall Structure of the Algorithms

1. Create start point node – push onto open list
2. While open list is not empty

   A. Pop node from open list (call it currentNode)

   B. If currentNode corresponds to goal, break from step 2

   C. Create new nodes (successors nodes) for cells around currentNode and push them onto open list

   D. Put currentNode onto closed list

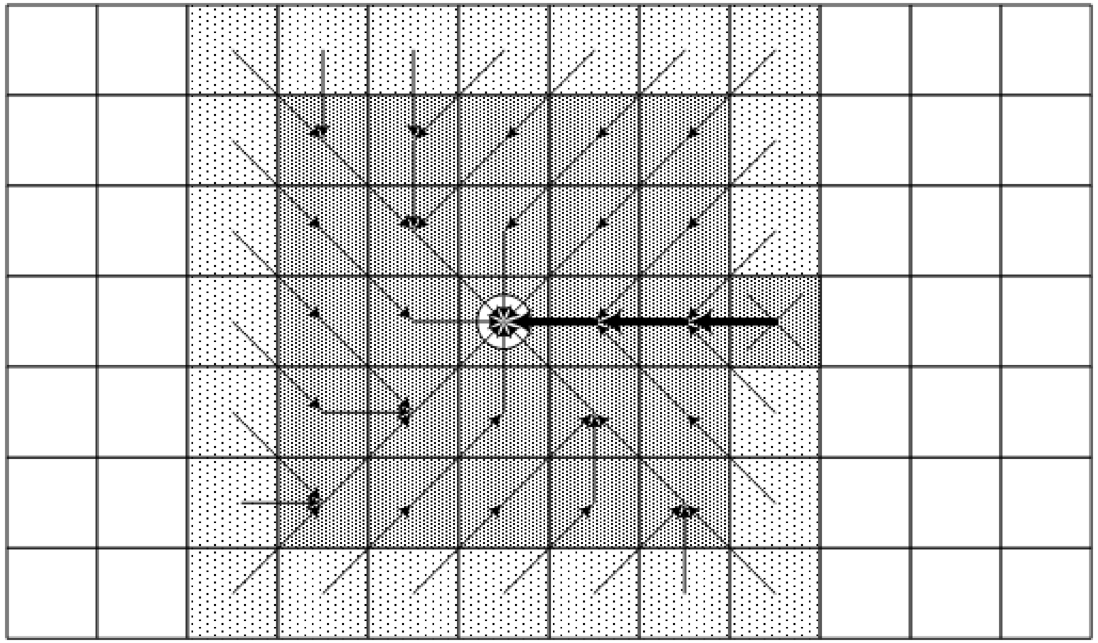
! when creating new node, make sure there is no more than 1 node for any given cell (otherwise multiple paths to same destination): need strategy to keep only 1 path for every cell

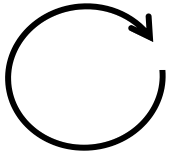
! difference between algorithms is how to choose which node from open list to process first

Breadth-First

- Finds a path from the start to the goal by examining the search space ply-by-ply

points to parent





Breadth-First Characteristics

- Exhaustive search
  - Systematic, but not clever; does not use position of goal to focus search
- Consumes substantial amount of CPU and memory
- Guarantees to find paths that have fewest number of nodes in them
  - Not necessarily the shortest distance!
- ***Complete*** algorithm
- implementation: use queue as data structure for open list (FIFO)

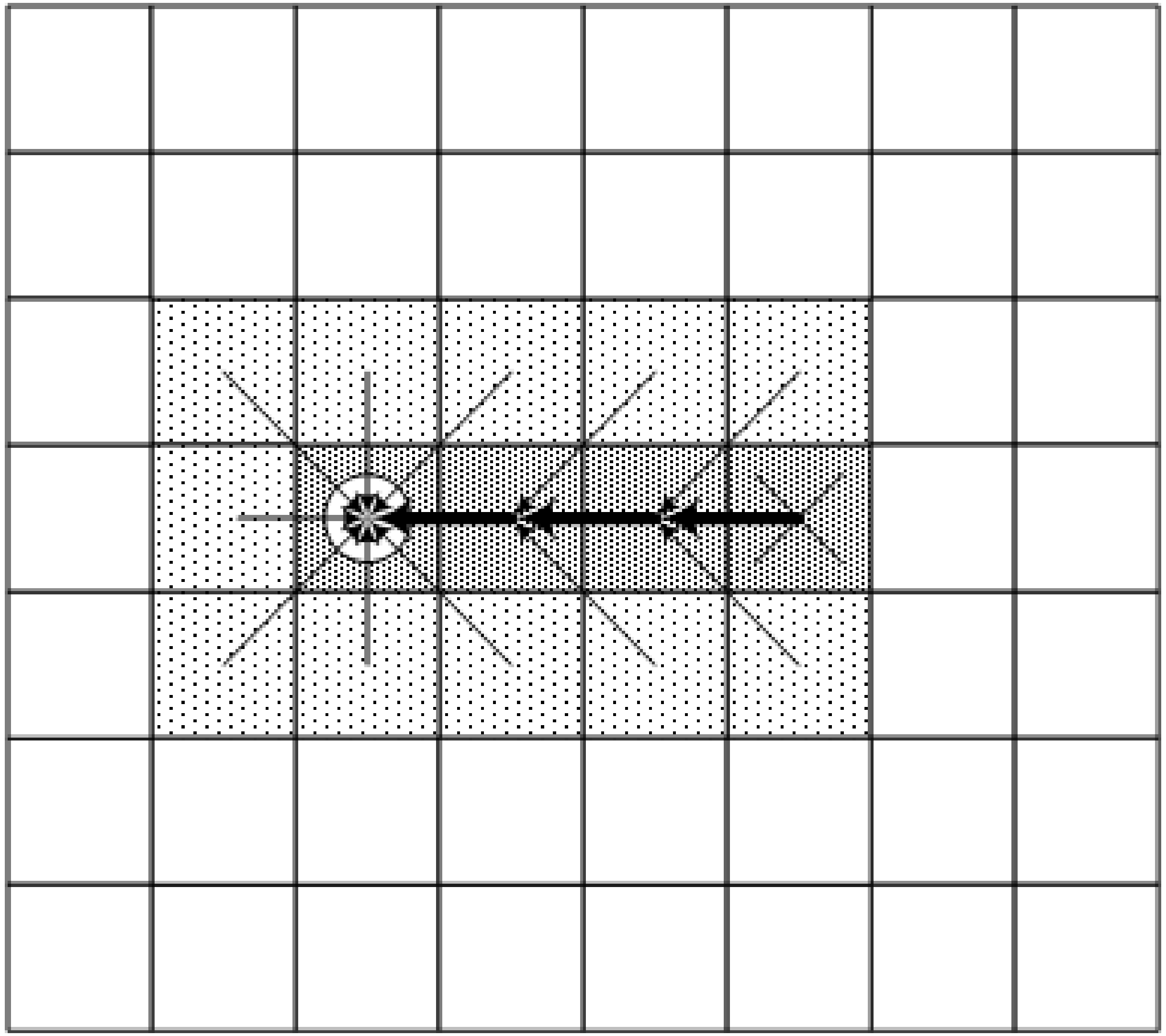
KU LEUVEN

Best-First

- Uses problem specific knowledge to speed up the search process
- Head straight for the goal
- Computes the distance of every node to the goal
  - Uses the distance (or heuristic cost) as a priority value to determine the next node that should be brought out of the open list
- Implementation: use priority queue as data structure for open list (smallest distance first)

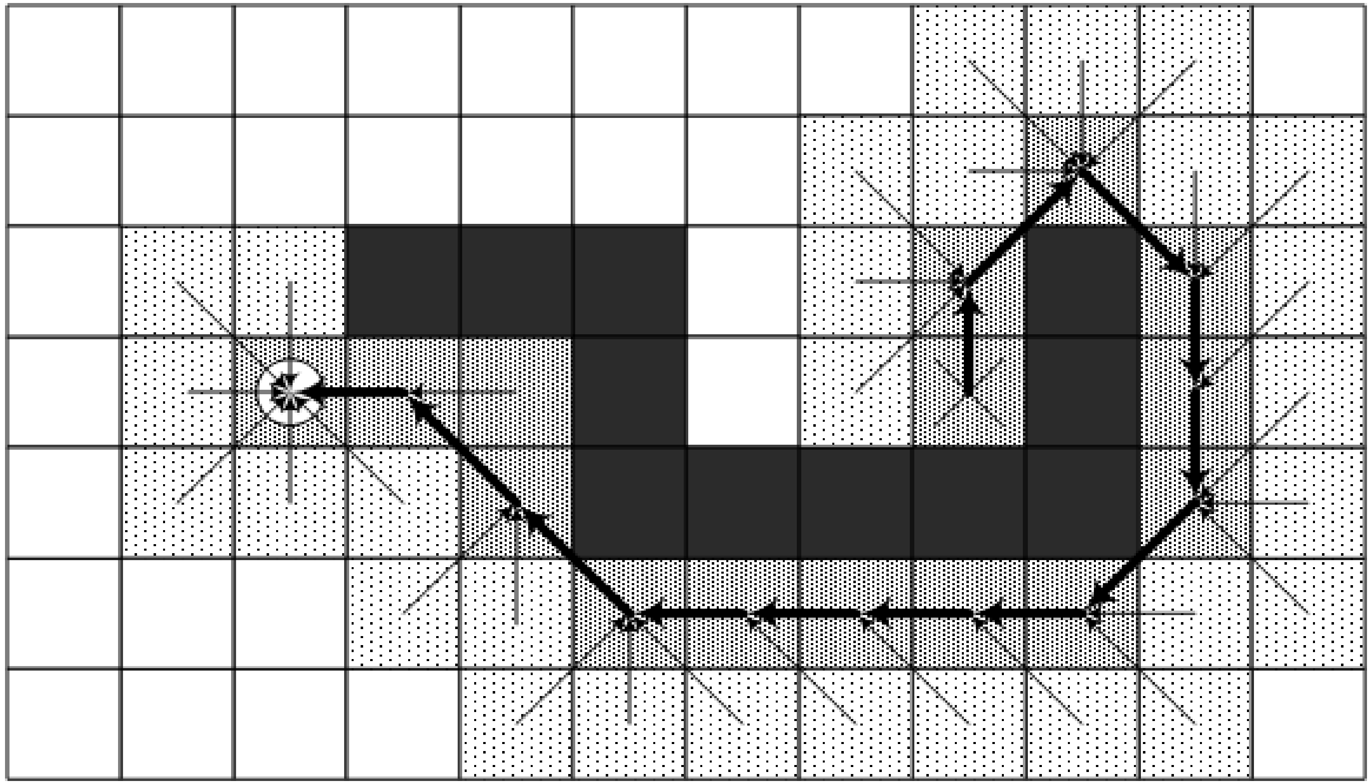
KU LEUVEN

Best-First (continued)



Best-First (continued)

- Situation where Best-First finds a suboptimal path



Best-First Characteristics

- Heuristic search
- Uses fewer resources than Breadth-First
- Tends to find good paths
  - No guarantee to find most optimal path
  - Distance is heuristic with weaknesses
- ***Complete*** algorithm

Dijkstra

- Disregards distance to goal
  - Keeps track of the cost of every path
  - No guessing
- Computes accumulated cost paid to reach a node from the start
  - Uses the cost (called the given cost) as a priority value to determine the next node that should be brought out of the open list

Dijkstra Characteristics

- Exhaustive search
- At least as resource intensive as Breadth-First
- Always finds the most optimal path
- ***Complete*** algorithm
- Flexibility in how to define cost, e.g. risk of being seen by enemy – can be adapted over time
- Used in lot of applications: network routing, routeplanners…

A*

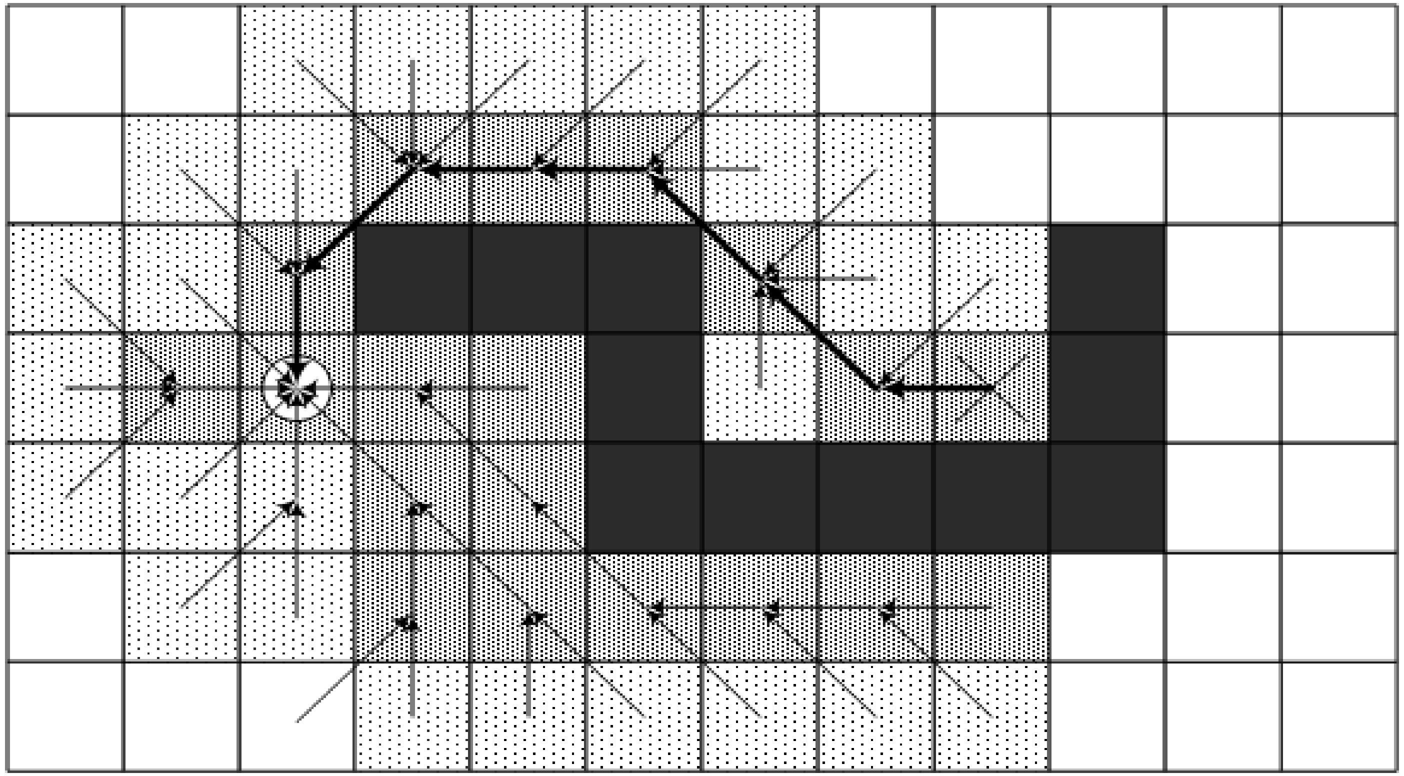
- Uses both heuristic cost and given cost to order the open list
- Final Cost = Given Cost + (Heuristic Cost * Heuristic Weight)
  - weight → ∞ : Best-first
  - weight = 0 : Dijkstra

That’s the slider you need in your GUI

KU LEUVEN

A* (continued)

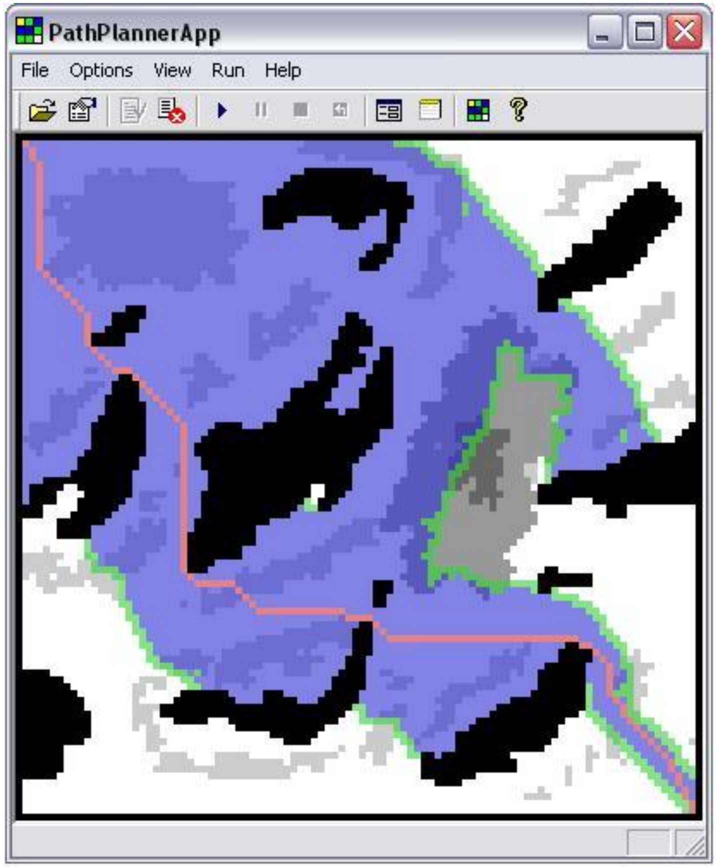
- Avoids Best-First trap!



A* Characteristics

- Heuristic search
- On average, uses fewer resources than Dijkstra and Breadth-First
- *Admissible* (never overestimate true cost, otherwise would leave optimal path) heuristic guarantees it will find the most optimal path: distance is OK!
- ***Complete*** algorithm

PathPlannerApp



Your implementation

- A good starting point is the manual of the PathPlannerApp (available on Toledo, most important parts marked in yellow)
- Discusses the most important aspects and guides you through the “difficult” spots
- Discusses optimalization
- But is an “old” document (some facts about STL are no longer valid)
- Data structures?
  - Is becoming a difficult question due to improved chaching strategies and increased (L3) cache size
  - First candidate is std::priority_queue for open list
  - Try to avoid searching in huge data structures… how?

KU LEUVEN