DM842

Computer Game Programming: AI

## Lecture 5
# Path Finding

Christian Kudahl

Department of Mathematics & Computer Science
University of Southern Denmark

# Outline

1. Heuristics

2. World Rerpresentations

3. Hierarchical Pathfinding

# Heuristics

Admissible (underestimating):

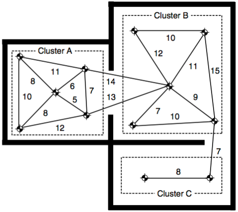
- has the nice properties of optimality
- more influence by cost-so-far
- increases the runtime, gets close to Dijkstra

Inadmissible (overestimating)

- less influence by cost-so-far
- if overestimate by $\epsilon$ then path at most $\epsilon$ worse
- in practice beliviability is more important than optimality

Common heuristics

- Euclidean heuristic (straght line without obstacles, underestimating) good in outdoor, bad in indoor
- Octile distance
- Cluster heuristic: group nodes together in clusters (eg, cliques) representing some highly interconnected region.
  Precompute lookup table with shortest path between all pairs of clusters.
  If nodes in same cluster then Euclidean distance else lookup table. Good for indoors. Knowledge vs Search Time

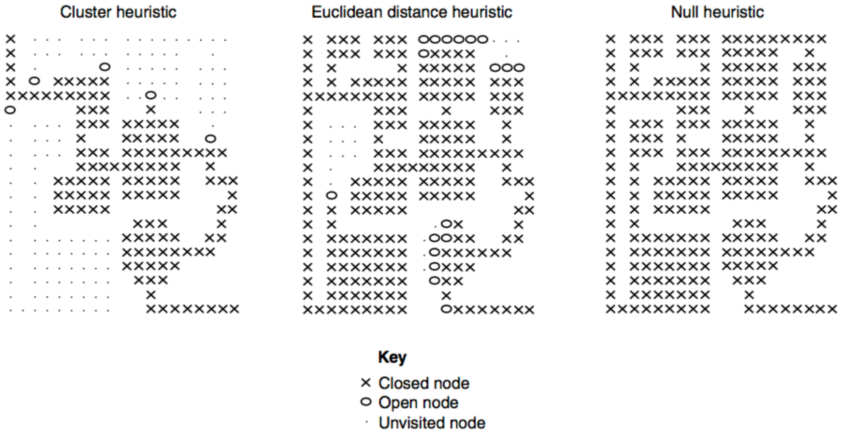



| | A | B | C |
|---|---|---|---|
| A | x | 13 | 29 |
| B | 13 | x | 7 |
| C | 29 | 7 | x |

Lookup table

Problems: all nodes of a cluster will have the same heuristic. Maybe add Euclidean heuristic in the cluster?

## Visualization of the fill



# Dominance

If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible)
then $h_2$ dominates $h_1$ and is better for search.

Given any admissible heuristics $h_a$, $h_b$,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates $h_a$, $h_b$

# Relaxed problems

Admissible heuristics can be derived from the *exact* solution cost of a *relaxed* version of the problem

- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution
- Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

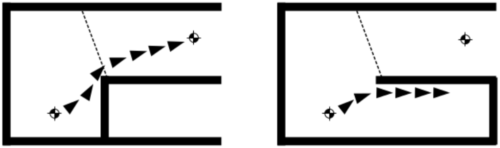
# Outline

1. Heuristics
2. World Rerpresentations
3. Hierarchical Pathfinding

# World Representations

Division scheme: the way the game level is divided up into linked regions that make the nodes and edges.
Properties of division schemes:

- quantization/localization
  from game world locations to graph nodes and viceversa
- generation
  how a continous space is split into regions
  *manual techniques*: Dirichlet domain
  *algorithmic techniques*: tile graphs, points of visibility, and navigation meshes
- validity
  all points in two connected reagions must be reachable from each other.

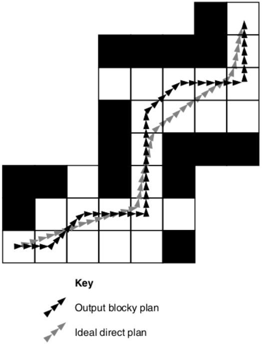


# Tile graphs

Division scheme:
Tile-based levels split world into regular square (or exagonal) regions.
(in 3D, for outdoor games graphs based on height and terrain data.)
Nodes represent tiles, connections with 8 neighboring tiles

Quantization (and Localization)
Each point is mapped in a tile by:

```
tileX = floor(x / tileSize)
tileZ = floor(z / tileSize)
```



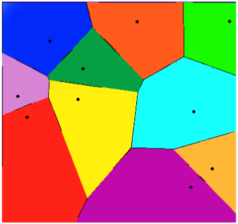

Generation:
automatic at run time, no need to store separately. Allow blocked tiles.

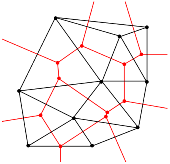
Validity:
partial bloackage could cause problems.

Remarks:
it may end up with large number of tiles
paths may look blocky and irregular

# Dirichlet Tassellation

Way of dividing space into a number of regions (aka Vornoi diagram)
A set of points (called seeds or sites) is specified beforehand.
For each seed there will be a corresponding region consisting of all points closer to that seed than to any other.

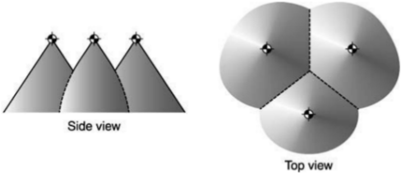




connecting circumcircles $\rightsquigarrow$ Vornoi decomposition

**Division scheme**:
Seeds (characteristic points) usually specified by a level designer as part of the level data
connections between bordering domains

Regions can be also left to define to the designer or cone representation and point of view.



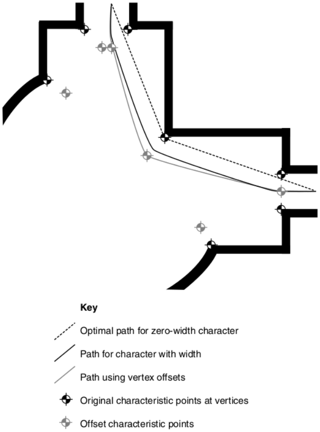

**Quantization**
find closest seed: use some kind of spatial partitioning data structure (ex $k$d-trees, as quad-tree, octree, binary space partition, or multi-resolution map)
**Validity**
May lead to invalid paths $\rightarrow$ Good seed placement makes it work in practice. Leave Obstacle and Wall Avoidance on.
**Advantage:** Moving the seeds, the pathfinding can be changed without changing the level itself.

# Points of Visibility

Inflection points: points on the path where the direction changes, may not be feasible for the character due to collision. Need to be moved.

Division scheme:
inflection points: Look at level geometry (maybe costly) or generate specially.
connection is made if the ray doesn't collide with any other geometry

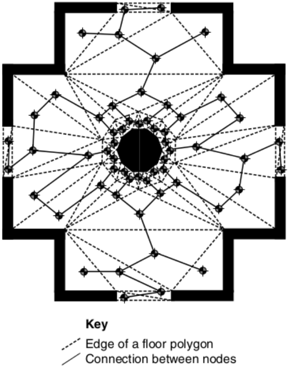
Quantization:
Points of visibility are usually taken to represent the centers of Dirichlet domains



# Navigation Meshes

Navmesh: Designer specifies the way the level is connected and the regions it has by defining the graphical structure made up of polygons connected to other polygons.

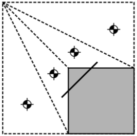
**Division scheme**:
floor polygons are nodes
connections if polygons share an edge



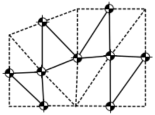

**Quantization and Localization**:
Coherence refers to the fact that, if we know which location a character was in at the previous frame, it is likely to be in the same node or an immediate neighbor on the next frame. Check first these nodes.
(note, polygons must be convex)

**Validity**:

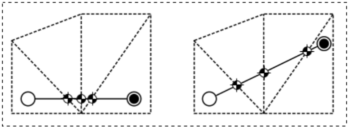
Not always guaranteed



Alternative division scheme: polygon-as-node vs edge-as-node
nodes on the edges between polygons and connections across the face of each polygon.



Nodes are sometimes allowed move on the edge. $\rightarrow$ expensive.

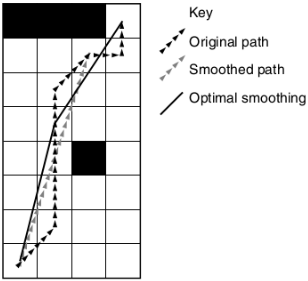


# Other Issues

- Cost maybe more than simple distance
- Different cost functions for different characters (tactical pathfinding)
- Tile-based graphs tend to be erratic.
  steering behaviours can take care of this.

# Path smoothing

```
def smoothPath(inputPath):
  if len(inputPath) == 2: return inputPath
  outputPath = [inputPath[0]]
  # We start at 2, because we assume two adjacent
  # nodes will pass the ray cast
  inputIndex = 2
  while inputIndex < len(inputPath)-1:
    if not rayClear(outputPath[len(outputPath)-1],
      inputPath[inputIndex]):
      outputPath += inputPath[inputIndex-1]
    inputIndex ++
  outputPath += inputPath[len(inputPath)-1]
  return outputPath
```




Note: output is a list of nodes that are in line of sight but among which we may have no connection

# Outline

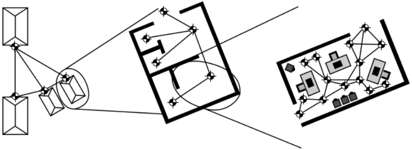
1. Heuristics

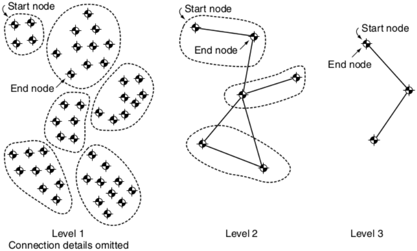
2. World Rerpresentations

3. Hierarchical Pathfinding

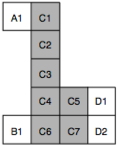
# Hierarchical Pathfinding

- multi-level plan: plan an overview route first and then refine it as needed.
- grouping locations together to form clusters.





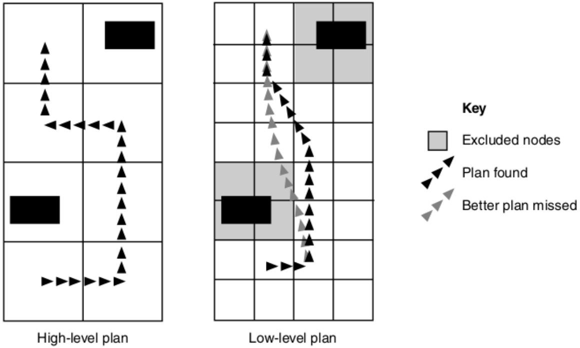

- edges between clusters that are connected
- costs not trivial: heuristics: minimum distance, maximin distance, average minimum distance



# Hierarchical Pathfinding

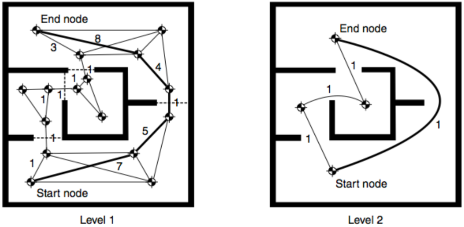
- apply A* algorithm several times, starting at a high level of the hierarchy and working down.
- results at higher levels used to limit the work at lower levels.
- end point is set at the end of the first move in the high-level plan.
- no need to initially know the fine detail of the end of the plan; we need that only when we get closer
- data structures: we need to convert nodes between different levels of the hierarchy.
  increasing the level of a node, simply find which higher level node it is mapped to.
  decreasing the level of a node, one node might map to any number of nodes at the next level down (localization). Choose representative point: center of nodes mapped to same node (easy geometric preprocessing), most connected node, etc.

Further speed-up:
Consider only nodes that are within the group that is part of the path, when refining at lower levels.



# Pathological cases

High-level pathfinding finds a route that can be a shortcut at a lower level.

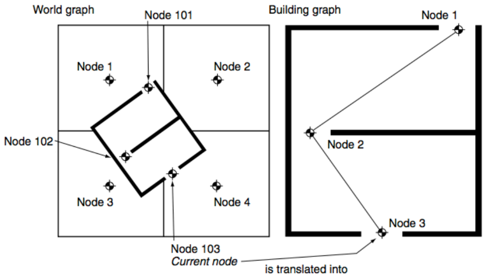



Minimum distance heuristic between rooms

Similar bad examples exist for the other cost functions.

# Instanced Geometry

- For each instance of a building in the game, keep a record of its type and which nodes in the main pathfinding graph each exit is attached to.
- Similarly, store a list of nodes in the main graph that should have connections into each exit node in the building graph.
- The instance graph acts as a translator. When asked for connections from a node, it translates the requested node into a node value understood by the building graph.



# Summary

- Best first search
    - Dijkstra
    - Greedy search
    - $A^*$ search
- Heuristics
- World representations
    - Tile graphs
    - Dirichelt tassellation
    - Points of visibility
    - Navigation meshes
    - Path smoothing
- Hierarchical Pathfinding

- Optimality
- Data structures