DM842
Computer Game Programming: AI

Lecture 5
Path Finding

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Pathfinding
2. Heuristics
3. World Rerpresentations
4. Hierarchical Pathfinding

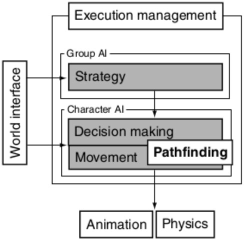
Motivation

For some characters, the route can be prefixed but more complex characters don't know in advance where they'll need to move.

- a unit in a real-time strategy game may be ordered to any point on the map by the player at any time
- a patrolling guard in a stealth game may need to move to its nearest alarm point to call for reinforcements,
- a platform game may require opponents to chase the player across a chasm using available platforms.

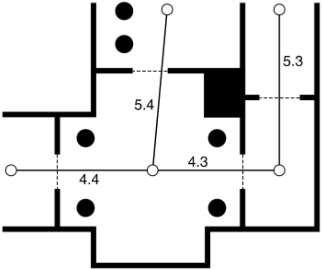
We'd like the route to be sensible and as short or rapid as possible

⇝ pathfinding (aka path planning) finds the way to a goal decided in decision making



Graph representation

Game level data simplified into directed non-negative weighted graph



node: region of the game level, such as a room, a section of corridor, a platform, or a small region of outdoor space

edge/arc: connections, they can be multiple

weight: time or distance between representative points or a combination thereof

Best first search

State Space Search
We assume:

- A start state
- A successor function
- A goal state or a goal test function

- Choose a metric of best
  Expand states in order from best to worst

- Requires:
  Sorted open list/priority queue
  closed list
  unvisited nodes

Best first search

Definitions

- Node is expanded/processed when taken off queue
- Node is generated/visited when put on queue
- g-cost is the cost from the start to the current node
- $c(a, b)$ is the edge cost between a and b

Algorithm Measures

- Complete
  Is it guaranteed to find a solution if one exists?
- Optimal
  Is it guaranteed the find the optimal solution?
- Time
- Space

Best-First Algorithms

Best-First Pseudo-Code

```
Put start on OPEN
While(OPEN is not empty)
  Pop best node n from OPEN # expand n
  if (n == goal) return path(n, goal)
  for each child of n: # generate children
    put/update value on OPEN/CLOSED
  put n in CLOSED
return NO PATH
```

Best-First child update

```
If child on OPEN, and new cost is less
  Update cost and parent pointer
If child on CLOSED, and new cost is less
  Update cost and parent pointer, move node
        to OPEN
Otherwise
  Add to OPEN list
```

Search Algorithms

Dijkstra's algorithm $\equiv$ **Uniform-Cost Search (UCS)**
$\rightsquigarrow$ Best-first with g-cost
Complete? Finite graphs yes, Infinite yes if $\exists$ finite cost path, eg, weights $> \epsilon$
Optimal? yes

Idea: reduce fill nodes: Heuristic: estimate of the cost from a given state to the goal

Pure Heuristic Search / **Greedy Best-first Search (GBFS)**
$\rightsquigarrow$ Best-first with h-cost
Complete? Only on finite graph
Optimal? No

A*
$\rightsquigarrow$ best-first with f-cost, $f = g + h$
Optimal? depends on heuristic

Termination

When the node in the open list with the smallest cost-so-far has a cost-so-far value greater than the cost of the path we found to the goal, ie, at expansion (like in Dijkstra)

Note: with any heuristic, when the goal node is the smallest estimated-total-cost node on the open list we are not done since a node that has the smallest estimated-total-cost value may later after being processed need its values revised.

In other terms: a node may need revision even if it is in the closed list ($\neq$ Dijkstra) because. We may have been excessively optimistic in its evaluation (or too pessimistic with the others).

(Some implementations may stop already when the goal is first visited, or expanded, but then not optimal)

However if the heuristic has some properties then we can stop earlier:

Theorem

If the heuristic is:

- admissible, i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n ($h(n) \geq 0$, so $h(G) = 0$ for any goal G)
- consistent

$$h(n) \leq c(n, n') + h(n') \qquad n' \text{ sucessor of } n$$

(triangular inequality holds)

then when A* selects a node for expansion (smallest estimated-total-cost), the optimal path to that node has been found.

E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

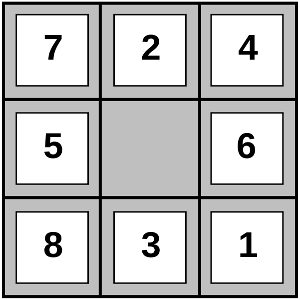
Note:

- consistent $\Rightarrow$ admissible
- if the graph is a tree, then admissible is enough.

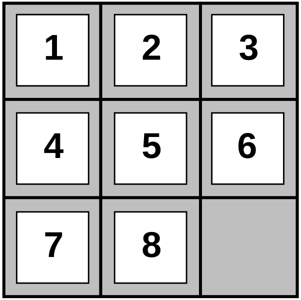
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$h_1(S) = 6$
$h_2(S) = 4+0+3+3+1+0+2+1 = 14$

Consistency

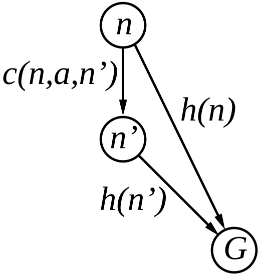
A heuristic is consistent if

$$h(n') \leq c(n, a, n') + h(n)$$

If h is consistent, we have

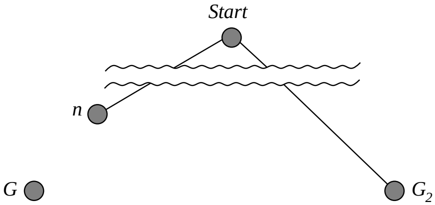
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.



Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue.
Let n be an unexpanded node on a shortest path to an optimal goal G_1.



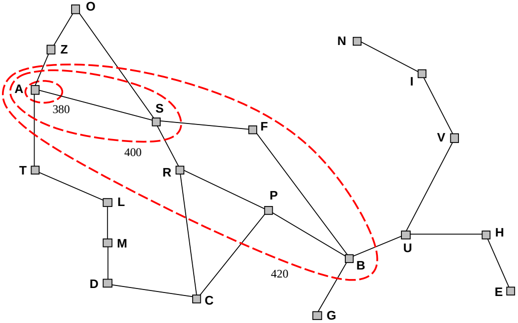
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Since $f(G_2) > f(n)$, A* will not select G_2 for expansion before reaching G_1

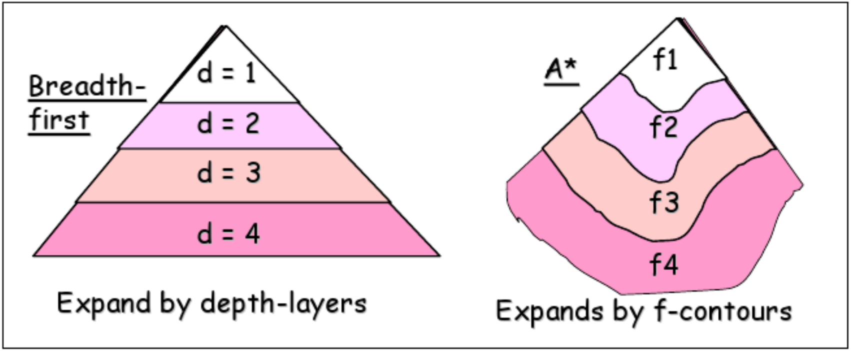
Optimality of A*

Lemma: A* expands nodes in order of increasing f value*
Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$.

And it does not expand any node n: $f(n) > c^*$



A* vs. Breadth First Search



Properties of A*

Complete? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Optimal? Yes—cannot expand f_{i+1} until f_i is finished
A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$
Time $O(lm)$, Exponential in [relative error in $h \times$ length of sol.]
l number of nodes whose total estimated-path-cost is less than that of the goal.
Space $O(lm)$ Keeps all nodes in memory

Data Structures

Same as Dijkstra:

list used to accumulate the final path: not crucial, basic linked list

graph : not critical: adjacency list, best if arcs are stored in contiguous memory, in order to reduce the chance of cache misses when scanning

open and closed lists: critical!

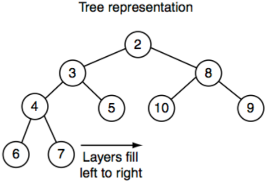
1. push
2. remove
3. extract min
4. find an entry

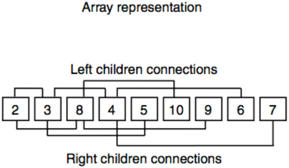
Priority queues
keep list sorted by finding right insertion point when adding.
If we use an array rather than a linked list, we can use a binary search

Priority heaps

- array-based data structure which represents a tree of elements.
- each node has up to two children, both with higher values.
- balanced and filled from left to right
- node i has children in positions $2i$ and $2i+1$

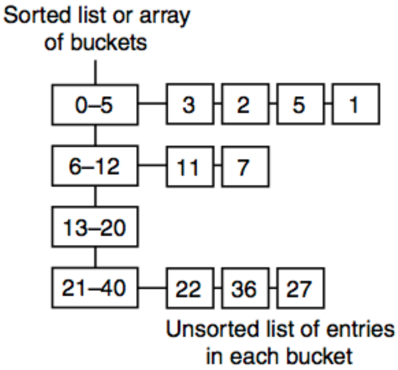
- extract min in $O(1)$
- adding $O(\log n)$
- find $O(\log n)$
- remove $O(\log n)$





Bucketed Priority Queues

- partially sorted data structure
- buckets are small lists that contain unsorted items within a specified range of values.
- buckets are sorted but their contents not
- exctract min: go to the first non-empty bucket and search its contents
- find, add and remove depend on number of buckets and can be tuned.
- extensions: multibuckets



Implementation Details

Data structures:

- author: depends on the size of the graph with million of nodes bucket priority list may outperform priority buffer
  But see `http://stegua.github.com/blog/2012/09/19/dijkstra/`

Heuristics:

- implemented as functions or class.
- receive a goal so no code duplication
- pathfindAStar(graph, start, **end**, new Heuristic(**end**))
- efficiency is critical for the time of pathfind
  Problem background, Pattern Databases, precomputed memory-based heuristic

Other:

- overall must be very fast, eg, 100ms split in 1ms per frame
- 10MB memory

Other heuristic speedups (Nathan Sturtevant)

- Break ties towards states with higher g-cost

- If a successor has f-cost as good as the front of OPEN
  Avoid the sorting operations

- Make sure heuristic matches problem representation
  With 8-connected grids don't use straight-line heuristic

- weighted A*: $f(n) = (1 - w)g(n) + wh(n)$

Node Array A^*

- Improvement of A* when nodes are numbered with sequential integers.
- Trade memory for speed
- Allocate array of pointers to records for all nodes of the graph. (many nodes will be not used)
- Thus Find in $O(1)$
- A field in the record indicates: unvisited, open, or closed
- Closed list can be removed
- Open list still needed

Outline

Heuristics

Admissible (underestimating):

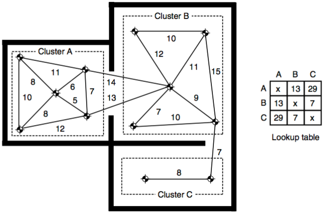
- has the nice properties of optimality
- more influence by cost-so-far
- increases the runtime, gets close to Dijkstra

Inadmissible (overestimating)

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

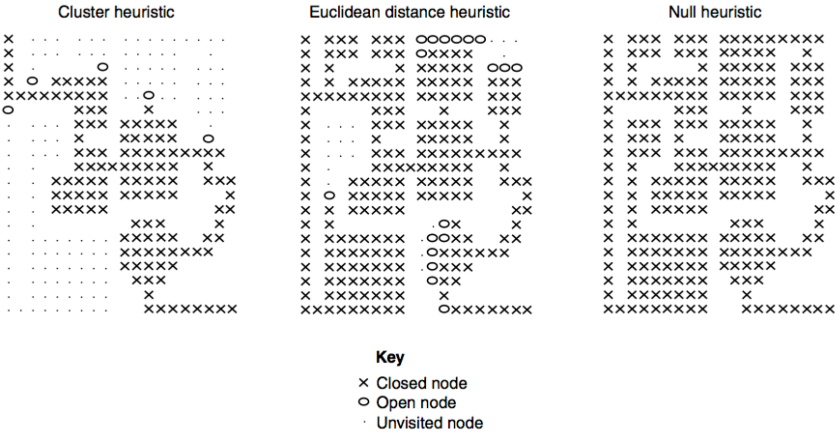
Common heuristics

- 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



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

Typical search costs:

$d = 14$	IDS = 3,473,941 nodes
	$A^*(h_1)$ = 539 nodes
	$A^*(h_2)$ = 113 nodes
$d = 24$	IDS $\approx$ 54,000,000,000 nodes
	$A^*(h_1)$ = 39,135 nodes
	$A^*(h_2)$ = 1,641 nodes

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

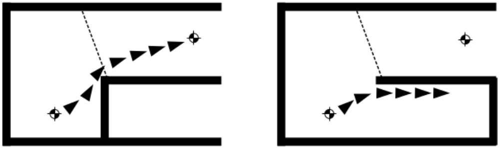
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution
- Key point: the optimal solution cost of a relaxed problem is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

Outline

World Representations

Division scheme: the way the game level is divided up into linked reagions 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.

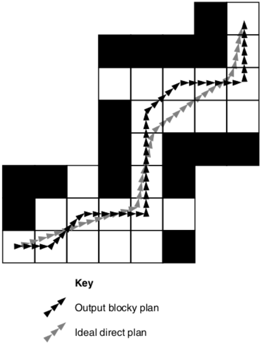


Tile graphs

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)
```



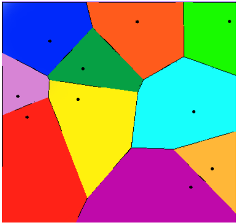
Generation:
automatic at run time, no need to store separately. Allow blocked tiles.

Validity:
with partial blockage might be not guaranteed.

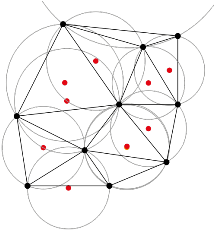
Remarks:
it may end up with large number of tiles
paths may look blocky and irregular

Dirichelet Tassellation

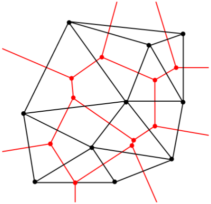
Way of dividing space into a number of regions (aka Vornoi diagram/decomposition)
A set of points (called seeds or sites) is specified beforehand.
For each seed there will be a corresponding region consisting of all points closer to that seed than to any other.



Dual of Delaunay triangulation



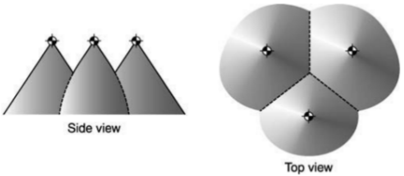
no point inside circumcircles of triangles (their centers in red).



connecting circumcircles ⇝ 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.
weighted Dirichlet domain: each point has an associated weight value that controls the size of its region.



Quantization
find closest seed: use some kind of spatial partitioning data structure (ex kd-trees, as quad-tree, octree, binary space partition, or multi-resolution map)

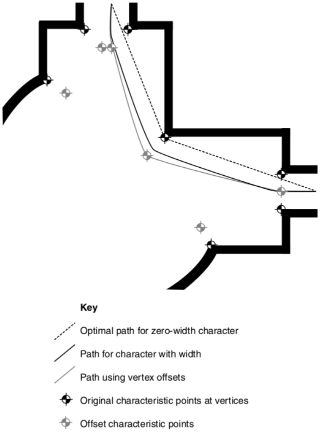
Validity
may lead to invalid paths. Leave Obstacle and Wall Avoidance on.

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

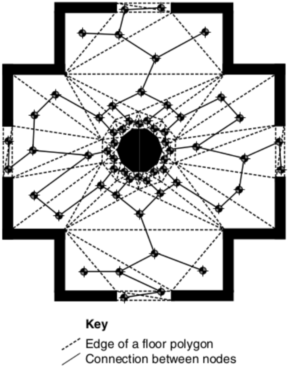
Quantization:
Points of visibility are usually taken to represent the centers of Dirichlet domains



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.

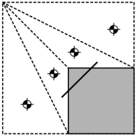
Division scheme:
floor polygons are nodes
connections if polygons share an edge



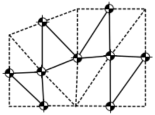
Quantization and Localization:
Coherence refers to the fact that, if we know which location a character was in at the previous frame, it is likely to be in the same node or an immediate neighbor on the next frame. Check first these nodes.
(note, polygons must be convex)

Validity:

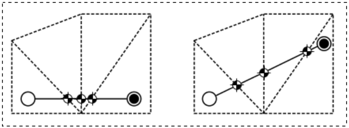
Not always guaranteed



Alternative division scheme: polygon-as-node vs edge-as-node
nodes on the edges between polygons and connections across the face of each polygon.
used in association with portal-based rendering, where nodes are assigned to portals and connections link portals on the same (convex) polygon.



Nodes may move on the edge.

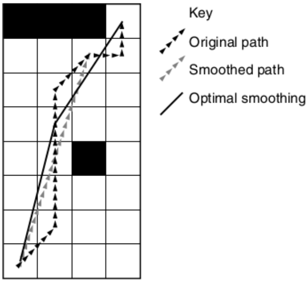


Other Issues

- Non-translational problems: nodes may indicate not only positions but also orientations
- Cost maybe more than simple distance
- Different cost functions for different characters (tactical pathfinding)
- Erratic paths
  portal representations with points of visibility tend to give smooth paths
  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
```

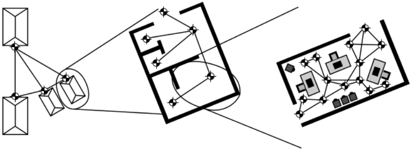


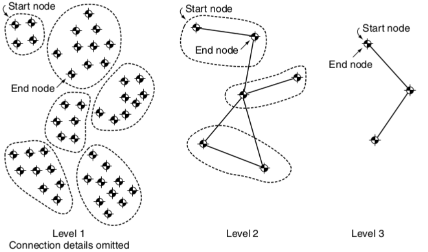
Note: output is a list of nodes that are in line of sight but among which we may have no connection

Outline

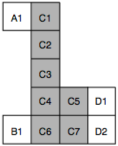
Hierarchical Pathfinding

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





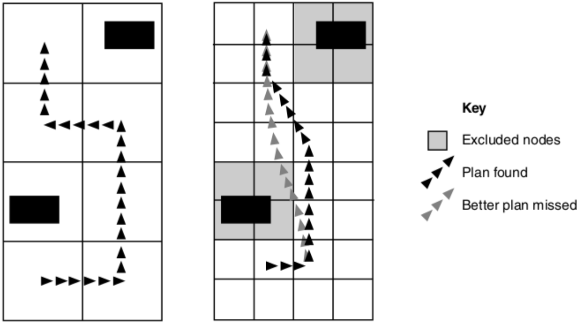
- edges between clusters that are connected
- costs not trivial: heuristics: minimum distance, maximum distance, average minimum distance



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.

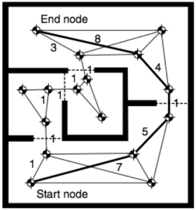


High-level plan

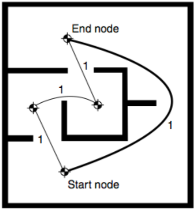
Low-level plan

Pathological cases

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

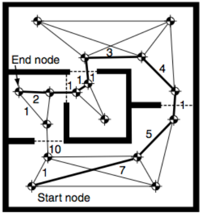


Level 1

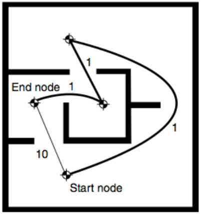


Level 2

Minimum distance heuristic between rooms



Level 1

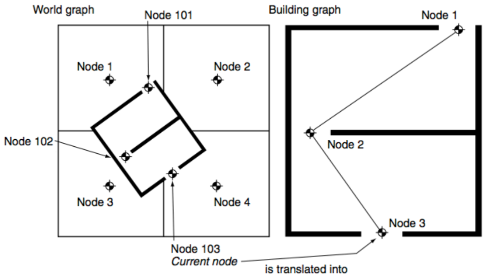


Level 2

Maximin distance heuristic between rooms

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.



Resume

- 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