DM810

Computer Game Programming II: AI

Lecture 7
Pathfinding
Decision Making

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Resume

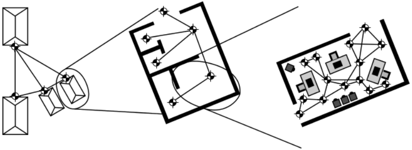
- Best first search
    - Dijkstra
    - Greedy search
    - A* search
- Heuristics
- World representations
    - Tile graphs
    - Dirichelt tassellation
    - Points of visibility
    - Navigation meshes
    - Path smoothing
- Hierarchical pathfinding
- Optimality
- Data structures

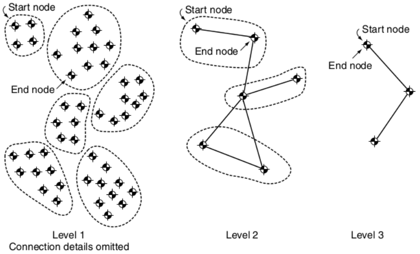
Outline

Outline

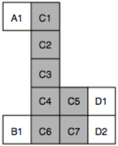
Hierarchical Pathfinding

- multi-level plan: plan an overview route first and then refine it as needed.
- we only need to plan the next part of the route when we complete a previous section.
- grouping locations together to form clusters.





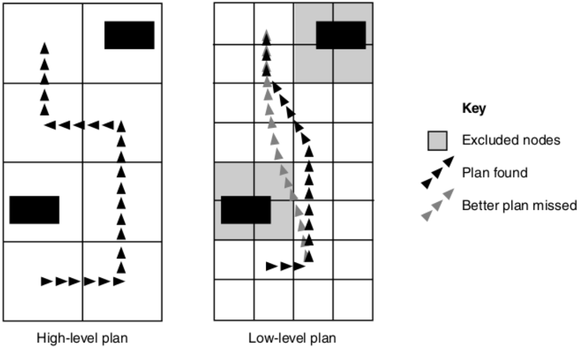
- edges between clusters that are connected
- costs not trivial: heuristics: minimum distance, maximum distance, average minimum distance



Hierarchical Pathfinding

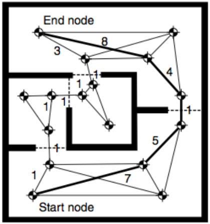
- apply A* algorithm several times, starting at a high level of the hierarchy and working down.
- results at higher levels used to limit the work at lower levels.
- end point is set at the end of the first move in the high-level plan.
- no need to initially know the fine detail of the end of the plan; 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.

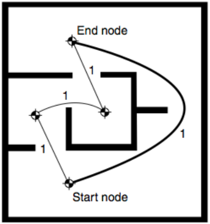


Pathological cases

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

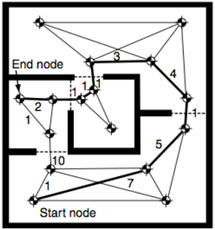


Level 1

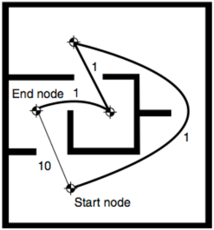


Level 2

Minimum distance heuristic between rooms



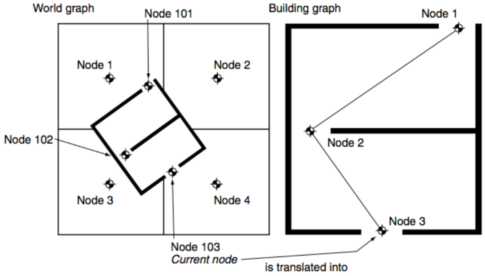
Level 1



Level 2

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.



Outline

Open Goal Pathfinding

- check if a node is a goal
- heuristics need to report the distance to the nearest goal.
  This is problematic and handled by decision making (selecting a goal).

Dynamic Pathfinding

- environment is changing in unpredictable ways or its information is incomplete.
- replan each time new information is collected
- replan only the part that has changed $\rightsquigarrow$ D* but requires a lot of storage space for, eg, storing path estimates and the parents of nodes in the open list

Memory-Bounded Search

- Try to reduce memory needs
- Take advantage of heuristic to improve performance
  - Iterative-deepening A* (IDA*)
  - SMA*

Iterative Deepening A*

- IDA*
  - Idea from classical Uniformed Iterative Deepening
    depth-first search where the max depth is iteratively increased
  - skip open and closed list
  - depth-first search with cutoff on the f-cost
  - cutoff set on the smallest f-cost of nodes that exceeded the threshold at the previous iteration
  - very simple to implement but less efficient
  - is the "best" variant for goal-oriented action planning in decision making

Properties of IDA*

Complete Yes
Time complexity Still exponential
Space complexity linear
Optimal Yes. Also optimal in the absence of monotonicity

Simple Memory-Bounded A*

Use all available memory

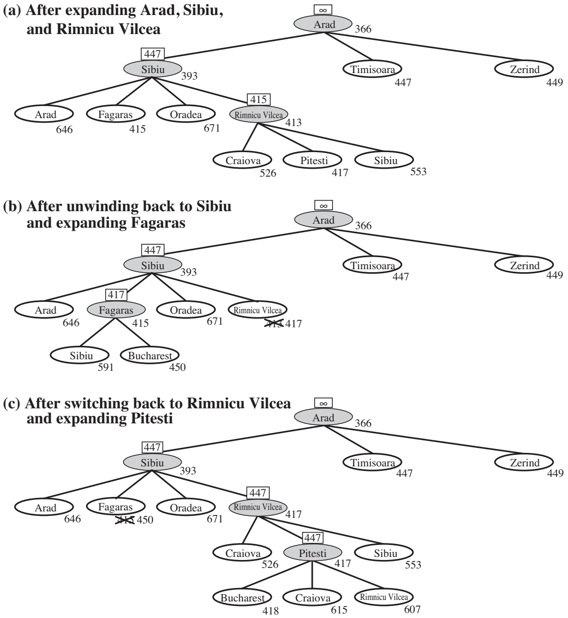
- Follow A* algorithm and fill memory with new expanded nodes
- If new node does not fit
  - remove stored node with worst f-value
  - propagate f-value of removed node to parent
- SMA* will regenerate a subtree only when it is needed
  the path through subtree is unknown, but cost is known

Properties of SMA*

Complete yes, if there is enough memory for the shortest solution path
Time same as A* if enough memory to store the tree
Space use available memory
Optimal yes, if enough memory to store the best solution path

In practice, often better than A* and IDA* trade-off between time and space requirements

Recursive Best First Search



Recursive Best First Search

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
     add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
     s.f ← max(s.g + s.h, node.f))
  loop do
     best ← the lowest f-value node in successors
     if best.f > f_limit then return failure, best.f
     alternative ← the second-lowest f-value among successors
     result, best.f ← RBFS(problem, best, min(f_limit, alternative))
     if result ≠ failure then return result
```

Other Issues

Interruptible Pathfinding

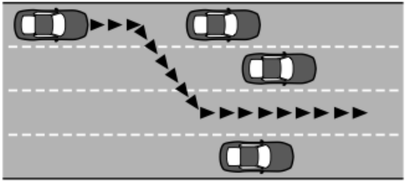
- needs to run every 60th or 30th of a second
- A* algorithm can be easily stopped and resumed.
- data required to resume are all contained in the open and closed lists.

In Real Time Strategy games: possible many requests to pathfinding at the same time

- serial $\rightsquigarrow$ problems for time, parallel $\rightsquigarrow$ problems for space
- pool of pathfinding + path finding queue (FIFO).
- information from previous pathfinding runs could be useful to be stored

Continuous Pathfinding

Vehicle pathfinding: eg, police car, path = a period of time in a sequence of lanes.



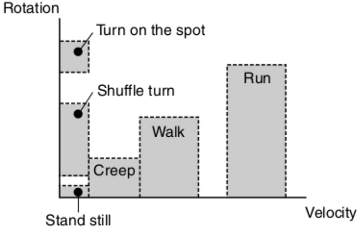
A discrete graph with fixed costs would not go. Other cars are moving. Depending on the speed the gap may be there or not.

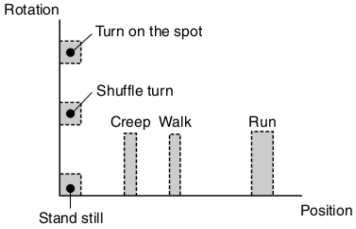
- A* in a graph where nodes represent states rather than positions
- a node has two elements: a position (made up of a lane and a distance along the road section) and a time.
- A connection exists between two nodes if the end node can be reached from the start node and if the time it takes to reach the node is correct.

- graph created dynamically: connections, so they are built from scratch when the outgoing connections are requested from the graph.
- retrieving the out-going connections from a node is a very time-consuming process $\rightsquigarrow$ avoid A^* versions that need recalculations
- It should be used for only small sections of planning.
  Eg, plan a route for only the next 100 yards or so. The remainder of the route planned on intersection-by-intersection basis.
  The pathfinding system that drove the car was hierarchical, with the continuous planner being the lowest level of the hierarchy.

Movement Planning

- If characters are highly constrained, then the steering behaviors might not produce sensible results. Eg: urban driving.
- Chars have, eg, walk animation, run animation, or sprint animation,





- Movement planning uses a graph representation. Each node of the graph represents both the position and the state of the character at that point, ie, the velocity vector, that determines the set of allowable animations that can follow

- Connections in the graph represent valid animations; lead to nodes representing the char after the animation

- route returned consists of a set of animations

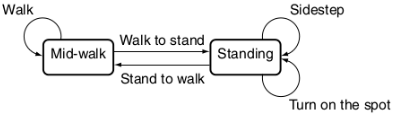
- If the velocities and positions are continuous, then infinite number of possible connections. Heuristic only returns the best successor nodes for addition to the open list.

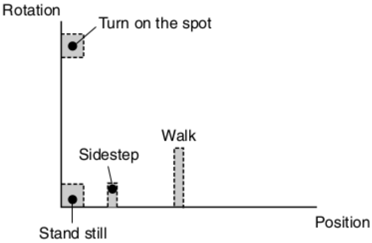
- similarly to continuous pathfinding, graph is generated on the fly and recomputations in A*are avoided.

Example

Walking bipedal character
Animations: walk, stand to walk, walk to stand, sidestep, and turn on the spot.
They can be applied to a range of movement distances
Positions: Each animation starts or ends from one of two positions: mid-walk or standing still.
Some positions in the environment are forbidden

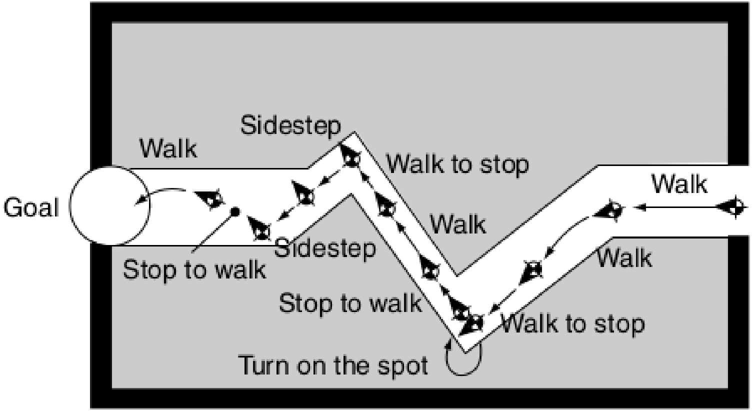
State machine: positions $\equiv$ states and transitions $\equiv$ animations.





Goal: range of positions with no orientation.

Result from A*:



Outline

1. Hierarchical Pathfinding

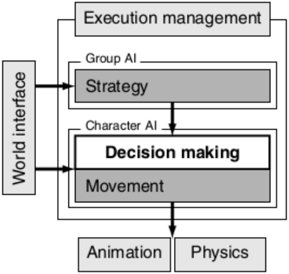
2. Other Ideas

3. Decision Making
   - Decision Trees

Decision Making

Decision Making: ability of a character to decide what to do.
We saw already how to carry out that decision (movement, animation, ...).
From animation control to complex strategic and tactical AI.

- state machines,
- decision trees
- rule-based systems
- fuzzy logic
- neural networks



Input internal and external knowledge
Output action

Knowledge representation:

- External knowledge identical for all algorithms
  Message passing system.
  Eg, `danger` is a constant at the character. Every new object in toolchain needs to define when to send message `danger` and the character will react.

- Internal knowledge algorithm dependent

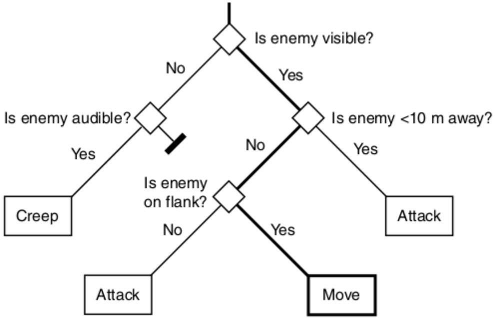
- Actions:
  Objects notify which actions they are capable of by means of flags.
  For goal oriented behavior, every action has a list of goals that will be achieved
  Alternatively, actions as objects with associated data such as state of world after action, animations, etc. Actions are then associated to objects.

The Toolchain

- AI-related elements of a complete toolchain
- Custom-designed level editing tools to be reused over all the games
- data driven or object oriented. Each object in the game world has a set of data associated with it that controls behavior
  Eg, data type "to be avoided" / "to be collected".
- Different characters require different decision making logic and behavior
- Allowing level designers to have access to the AI of characters they are placing without a programmer requires specialist AI design tools.
  Eg: AI-Implant and SimBionic provide a palette of AI behaviour to combine
- Actions selected by level designer are mostly steering behaviors.
  They are put together by the graphical definition of finite state machines
- Debugging at run time
- SDK that allows new functionality to be implemented in the form of plug-in tools.

Decision Trees

- Tree made up of connected decision points.
- Each choice is made based on the character's knowledge.
- At each leaf of the tree an action is attached
- Typically binary tree (multibranches are equivalent) but more generally directed acyclic graph (DAG).



Data Type	Decisions
Boolean	Value is true
Enumeration (i.e., a set of values,only one of which might be allowable)	Matches one of a given set of values
Numeric value (either integer or floating point)	Value is within a given range
3D Vector	Vector has a length within a given range (this can be used to check the distance between the character and an enemy, for example)

Combinations of decisions are obtained by the structure of the tree. Eg: AND, OR

Decision trees can express any function of the input attributes.
E.g., for Boolean functions, truth table row path to leaf

Execution time depends on decisions
Eg, checking if any enemy is visible may involve complex ray casting sight checks through the level geometry.

Implementation

A simple tree can be implemented initially, and then as the AI is tested in the game, additional decisions can be added.

```
class DecisionTreeNode:
  def makeDecision() # Recursion

class Action: #interfacing virtual functions
  def makeDecision():
  return this

class Decision (DecisionTreeNode):
  trueNode # pointer to a node
  falseNode
  testValue # pointer to data for the test
  def getBranch() # carries out the test
  def makeDecision() # Recursion

class FloatDecision (Decision):
  minValue
  maxValue
  def getBranch():
  if maxValue >= testValue >= minValue:
    return trueNode
  else:
    return falseNode
```

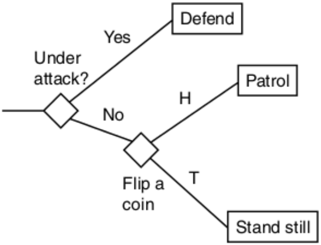
```
class MultiDecision (DecisionTreeNode):
  daughterNodes
  testValue

  def getBranch():
    return daughterNodes[testValue]

  def makeDecision():
    branch = getBranch()
    return branch.makeDecision()
```

Random Decision Trees

- Some element of random behavior choice adds unpredictability, interest, and variation
- Requires some care if the choice is made at every frame to yield stable behavior $\rightsquigarrow$ keep track of last decision




```
struct RandomDecision (Decision):
  lastFrame = -1
  lastDecision = false
  def test():
    if frame() > lastFrame + 1: # old
      # Make a new decision
      lastDecision = randomBoolean()
    lastFrame = frame() # curr. frame num.
    return lastDecision
```

- Add a time-out information, so the agent changes behavior occasionally.