AI and Games

Module IAS

Anne-Gwenn Bosser

2017-2018

Outline

Computer Games and AI

Computer Games and AI

Characterising a Computer Game

Characteristics of a Computer Game

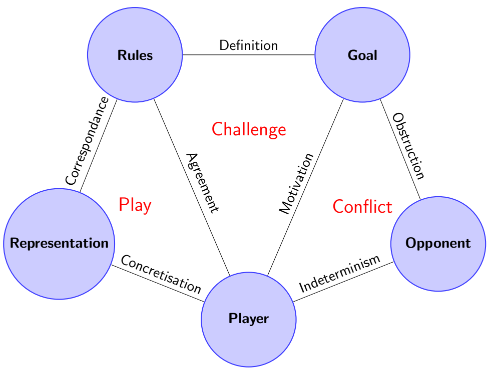
In groups of 2-3.
Think of 5 games that you know, computer games or not. Try to make the list of everything they have in common.

Characteristics of a Computer Game

Think of 5 games that you know, computer games or not. Try to make the list of everything they have in common.

1. Players
2. Rules
3. Goals
4. Opposing forces
5. Representation of the game in the *real* world

Play, Challenge and Conflict: Aspects of a Game



From (Smed and Hakonen 2005b)

Play, Challenge and Conflict: Aspects of a Game

From (Smed and Hakonen 2005b)

Challenge Rules **define** the game and its goal. Players **agree** to follow the rules. The goal **motivates** players.

Conflict Opposition (synthetic players whose behaviour is **unpredictible** from the player's point of view, or other human players) **obstructs** from achieving goal.

Play The rules are abstract but correspond to real-world objects (computer representation, board, haptics...) which concretises the game for human players.

AI can either support the player or oppose the player. It can be conceptualised as a *Synthetic player* which interacts with the game world.

Computer Games and AI

The Synthetic Player AI

The Synthetic Player AI: Chess

The computer program Deep Blue won the world champion Gary Kasparov in 1997 for the first time.

The Synthetic Player AI: Go

- Considered an intractable problem for a long time.
- Monte Carlo probabilistic methods from 2006 provided advances, but still to be used on supercomputers.
- The program MoGo first won a professional player in even non-blitz on 9x9 Goban in 2009.

The Synthetic Player AI: AlphaGo

AlphaGo has been developped by (Google) DeepMind (team now disbanded).

- Monte Carlo tree search + Deep Neural Networks + Reinforcement Learning
- Won 4-1 against a 9^{th} dan professional player Lee Sedol in March 2016.
- "Trained" online against professional players during 2016-2017 (won...)
- Won 3-0 against best-ranked human player Ke Jie in May 2017.

The Synthetic Player AI: Connect 4

This game is strongly solved: an algorithm can always find the best move for the player in a given situation (ex: minimax). With Perfect Play, the first player always win.

Perfect Play vs Artificial Stupidity?

Would you *enjoy* playing Connect 4 against a computer?

Perfect Play vs Artificial Stupidity?

Would you *enjoy* playing Connect 4 against a computer?
→ A game which is too difficult is not fun. How to formalise these notions?

To watch on your own time, reviews:

- In French, *Le joueur du grenier*: Tortue Ninja II on NES
- In English, *AVGN*: Ghosts n' Goblins

Flow theory: Optimal Experience (Csikszentmihalyi, 1975)

"A sense of that **oneÂôs skills are adequate to cope with the challenges at hand** *[...]. Concentration is so intense that there is* **no attention left over to think about anything irrelevant** *or to worry about problems. Self-consciousness disappears, and* **the sense of time becomes distorted***. An activity that produces such experiences is so* **gratifying** *that people are willing to do it for its own sake[...]"*

Read: Flow in games (and everything else), J. Chen, Com. ACM 2007 Viewpoints How does it feel to be in the flow?

- sense of novelty, achievement in mastering the unexpected
- pleasure in the activity itself rather than in reaching a goal alone

The 8 Dimensions of Flow (Csikszentmihalyi 93)

1	Clear goals and immediate feedback
2	Equilibrium between the level of challenge and personal skill
3	Merging of action and awareness
4	Focussed concentration
5	Sense of potential control
6	Loss of self-consciousness
7	Time distortion
8	Autotelic or self-rewarding experience

Not all of them have to be experienced to characterise this state of mind.

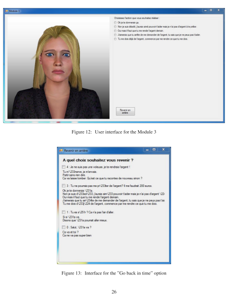
Applying Flow Theory to Games? Adaptation to the player

- Different players have different flow zones
- Games (try to) adapt player's flow experience through the choices deliberately built in the gameplay.
- Dynamic Difficulty Adjustment is the ability for a game to detect and adapt to an(y) individual's Flow experience. This is an area of research for AI.

Read: Flow in games (and everything else), J. Chen, Com. ACM 2007 Viewpoints

Towards Computational Intelligence

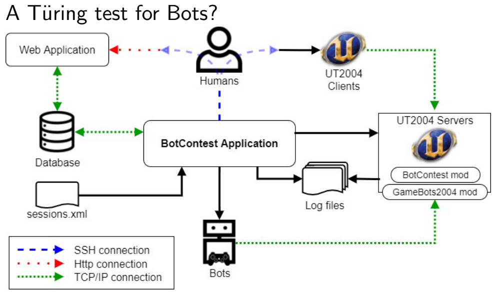
- *Intelligence* vs *Humanness*;
- The link between Artificial Intelligence and Virtual Reality
- Technologies: Virtual Agents
- Key Concepts: Co-presence, Emotional Computing



Even et al., 2015: Therapeutic game based on narrative generation techniques for Social Skills Rehabilitation. Reproduced with permission.

A big Issue: evaluating Believability

A Türing test for Bots?



Even et al., 2017: *Development of an assessment protocol for the believability of bots in video games. Reproduced with permission.*

Virtual Actors : is it just the AI?

Non Player Characters, Virtual Actors

- Multiplayer games: real human players
- Emotional computing: computational models of emotions, personality
- Intelligence, Humanity, Believability, ..., ?
- Dynamicity, real-time vs CGI designed cinematics.

Computer Games and AI

AI Based Games

AI Based Games: Simulation, Artificial Life, God Games

Neural Networks

- BDI, decision trees
- Reinforcement learning

Computer Games and AI

Other uses of AI for Games

Other possible use of aI for game applications

- dramatic tension metrics (adjustment of sounds, colors...) or management
- procedural content generation (terrain, narrative generation, sudoku, music ...)
- Intelligent Interfaces (coupled with user cognitive models), multimodal fusion, camera positioning

Path Planning

Path Planning

Introduction

Non-Player Characters AI (Agent based AI)

AI can help to provide NPCs with a level of autonomy, either for supporting or opposing the player.
Various levels of decision-making:

- Operational: should be cheap, real-time/reactive (ex: movement, get the ball, pass).
- Tactics: average cost to compute, regular (ex: cooperation/coordination between entities, keeping close to one particular opponent)
- Strategy: costly to compute, infrequent (ex: long term and speculative in nature, based on large amount of data, terrain analysis, risk analysis: defensive play, offensive play...)

First this week: path finding. Next: planning behaviours.

Game AI as a Synthetic Player

AI in Game World vs AI in Real World

video

AI in Game World vs AI in Real World

video

How can a bot find its path

1. Knowledge of the environment
2. Search algorithms
3. Believability

Pathfinding: principles

The problem: given a starting point S and a destination D, find the path leading from S to D taking into account a *cost* to minimize (usually travelling time).

1. Continuous search space: too costly → discretisation into a finite set of *enough* interconnected waypoints.
2. Given the waypoints closest to S and D, search the graph made of:
   - Vertices = Waypoints
   - Edges = Connections between waypoints
   - Weigth of edges = Costs (typically distance)

   for the cheapest path.
3. Animate the corresponding movement in the game world.

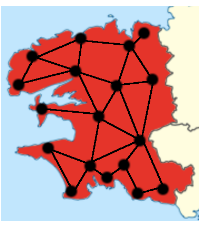
Path Planning

Discretisation of the Game World

Discretisation of the Game World

Can be defined:

- manually, during level-design (doorways, center of the rooms, corners,. . .)
- automatically, by the program

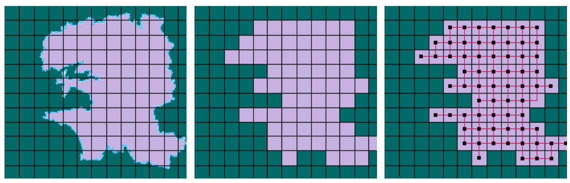


Automatic discretisation can be structured in a variety of ways, including:

- squares grid,
- hexagons grid,
- navmesh, ...

Automatic discretisation: Square Grid

- a tiling of polygons (*tessalation*) is laid over the game world for approximating the open game space.
- Waypoints: center of tiles; connections: neighbouring tiles' waypoints.



(*4- connectivity in square grid.*)

Variants

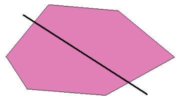
- Regular tessellations (made of identical regular tiles - of same angles and edge length) can also be made of equilateral triangles or hexagons.
- Waypoints can be affected to corners and connections to edges of the shapes.

Automatic construction of a Navigation Mesh

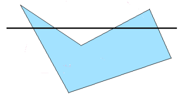
Definition

A convex polygon is a polygon with all its angles less than or equal to 180 °.

- Example: a triangle, a square (all *regular* tiles).
- A Property: a line which is not an edge drawn through the polygon will intersect with the polygon twice exactly.
- *Corollary: a NPC can always move in a straight line from one point to another inside a room shaped as a convex polygon.*



Convex polygon



Concave polygon

Navigation Mesh

Definition

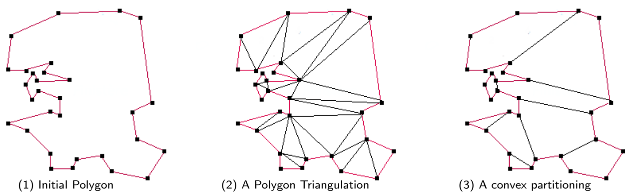
A Navigation mesh is a convex partitioning of the game world.

- Two adjacent polygons share one edge and two points
- No polygon overlaps.
- Each polygon represents a waypoint connected to adjacent polygons
- *Convexity guarantees that one can move in a straight line from any point within a polygon to any edge of the polygon, and from an edge to another* (e.g. NPC won't get stuck behind a wall or in a corner of the building).

Automatic Convex Partitioning

Example: the Hertel-Mehlhorn method.

- Use an automatic triangulation algorithm.
- Considering edges one by one, remove un-essential edges (which do not divide a concave angle).

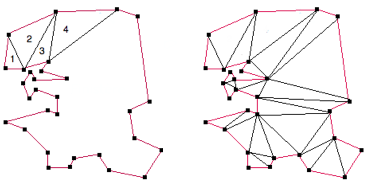


(1) Initial Polygon (2) A Polygon Triangulation (3) A convex partitioning

Polygon Triangulation

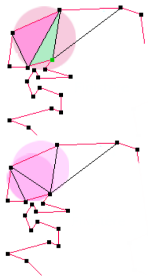
Ear-clipping: a triangulation method
Clip one *ear* of the polygon at a time

- Make a triangle from 2 adjacent edges of the polygon
- proceed iteratively with the rest, considering the created edge as the new polygon edge.



Polygon Triangulation

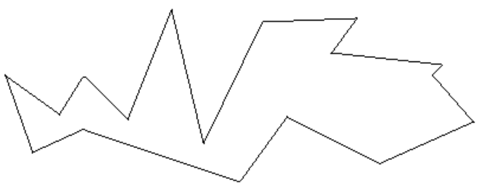
Definition
A triangulation is a Delaunay's triangulation if and only if for each triangle, the circumcircle does not contain any other vertice.



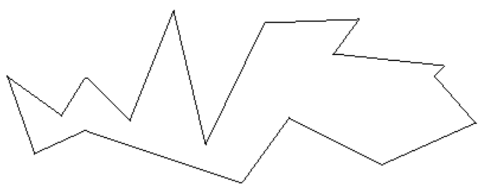
- algorithms creating *Delaunay's triangulation* will avoid "long pointy" triangles in the partition.
- A Delaunay's triangulation guarantees that between two vertices of the partition, the distance will be at most $4\pi/3\sqrt{3}$.

Some Practice

Triangulate and apply the Hertel-Mehlorn algorithm on the polygon below:



Some Practice

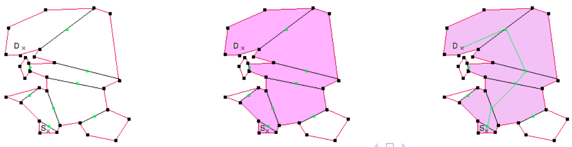


Pathfinding using a Navmesh

Given a starting point S and destination D:

1. Using polygons as waypoints, find the path from the polygon containing S to the polygon containing D.
2. Trace a line from S to the edge with the next polygon.
3. Trace a line between the entrance and leaving edges for each consecutive polygon.
4. Trace a line between the last edge and D.

Here centers of edges have been used as *waypoints*



Automatic Convex Partitioning Algorithms

With n the number of vertices of the initial polygon and r the number of concave angle vertices, complexity of a few commonly used algorithms:

- Green's optimal (minimising the number of polygons) partitioning algorithm (83): $\mathcal{O}(n^4)$ and $\mathcal{O}(n^3)$ space.
- Keil's optimal algorithm (85): $\mathcal{O}(r^2 n \; log \; n)$.
- Hertel-Melhorn heuristic (85) for a partition of at most 4 times the number of polygons in the optimal partition: $\mathcal{O}(n + r \; log \; r)$.

Path Planning

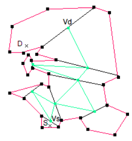
Finding the Path

Finding the Path

Path Finding as Search in State-Space

We follow the approach and terminology by [Russel and Norvig, 2010]

The discretised game world can now be treated as a graph, with vertices (waypoints), edges (connections), and where a cost such as travelling time can be associated as the edge's weight.



Defining the problem in terms of generic state space search:

- Initial State: In Vertice closer to the starting position S: $In(V_s)$
- Possible Actions: Follow one edge (to a neighbour vertice): $F(E_{V_i,V_j})$
- Transition Model: $In(V_i), F(E_{V_i,V_j}) \implies In(V_j)$
- Goal test: tests wether $In(V_d)$ (i.e we have reached the Vertice closer to the destination D).
- Path cost function: sum of weights of edges on the path.

Path Finding as Search in State-Space

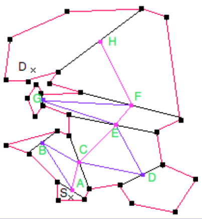
problem	initial state, possible actions, transition model, goal-reached test, cost function
solution	an action-sequence
optimal solution	the cheapest solution (wrt. the cost function)

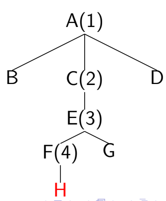
Definition

Expanding a state/node: applying each legal action from this state/state of the node, thus reaching new states/ creating children nodes.

Path Finding as Search in State-Space

- Working from the initial state, we construct a **search tree** by incrementally **expanding** considered states into children nodes.
- The set of all leaf nodes in the tree that can be expanded is called the **frontier**. Different search strategies will be used to select which of the frontier states will be expanded next.
- Because each child node knows its antecedent, a node verifying the goal gives a path/solution.





Path Finding as Search in State-Space

function GRAPH SEARCH(problem) **return** solution, failure
  frontier: Node built from initial state
  explored set: $\emptyset$
  loop
    if *frontier* $== \emptyset$ **then return** failure
    Choose and remove a node N from the frontier
    if the state in N verifies the goal test **then**
      return corresponding solution
    Add N to the explored set
    Expand N into $N_i{}_{i=1}^{n}$
    for i=1 to n **do**
      if $N_i \notin$ frontier and $N_i \notin$ explored set **then**
        add N_i to the frontier

Search strategies differ in how they choose the next Vertice to process in the frontier.

Search Infrastructure: what in a Node?

For each Node of the constructed search tree, the implementation defines:

- the corresponding Vertice in the graph: *a selected vertice* corresponds to a state in the state space we search.
- the parent node;
- the action which led from the parent to the current node (walk, run... one edge);
- a path-cost.

Search Infrastructure: Processing Nodes

Obtaining a child node given a parent node and possible action:

function CHILD-NODE(problem, parent, action) **return** a node
    return a node with
    State = problem.Result(parent.State, action)
    Parent = parent
    Action = action
    Pcost = parent.Pcost + problem.StepCost(parent.State, action)

Data Structures:

- Frontier: implement using a Queue: pop, empty, insert. Depending on the search strategy you will use either a LIFO (Stack), FIFO or a prioritised queue.
- Explored Set: Hashtable for constant time access.

Search Strategy Performance

- **Completeness**: is the algorithm guaranteed to find a solution?
- **Optimality**: does the strategy finds the optimal solution?
- **Time Complexity**: How long does it take to find a solution? (number of nodes generated)
- **Space Complexity**: How much memory is needed to perform the search? (max. number of nodes stored)

Complexity will be expressed either in terms of

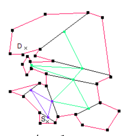
- number of Vertices and Edges (explicit/finite graph)
- branching factor, depth of shallowest goal, and maximum length of any path

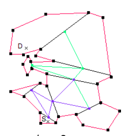
Classical tradeoff: balance between time and space complexity.
Another tradeoff: optimality of the solution vs. time to compute.

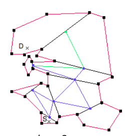
Breadth First Search

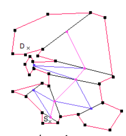
Search Principles: expanding root, then all direct descendants, then all their direct descendants...

- if *Start* = *Destination* then the path is found. If not proceed to the following step with $k = 1$;
- expand all paths to the set of vertices at distance k from *Start*. If *Destination* is not among these vertices, repeat with $k = k + 1$;
- if *Destination* is among these nodes then the shortest path has been found.


$k = 1$


$k = 2$


$k = 3$


$k = 4$

BFS: Pseudo Code

Implementation with the generic graph search technique:

- Use a FIFO Queue;
- test the Goal on generation.

BFS: Pseudo Code

function BREADTH-FIRST-SEARCH(problem) **return** solution, failure
  node := *a node with* State=problem.InitialState, PathCost = 0
  if problem.GOAL-TEST(node.State) **then return** SOLUTION(node)
  frontier := *a FIFO containing only* node
  explored := $\emptyset$
  loop
    if EMPTY(frontier) **then return** failure
    node := POP(**frontier**)
    add node.State *to* **explored**
    for all action **in** problem.ACTIONS(node.State) **do**
      child := CHILD-NODE(problem,node,action)
      if child.State $\notin$ explored*or* frontier **then**
        if problem.GOAL-TEST(CHILD.STATE) **then**
          return SOLUTION(child)
        frontier := INSERT(CHILD, FRONTIER)

Some Practice: Construct the BFS search tree

Construct a tree and number the nodes in the order of exploration: problem is to go from Arad to Bucarest.

BFS Performance

- Completeness: tick
- Optimality: if step costs are equal.
- Time Complexity: appalling $\mathcal{O}(b^{d+1})$
- Space Complexity: unmanageable $\mathcal{O}(b^d)$

Depth	Nodes	Time	Memory
2	110	.11ms	107KB
4	11110	11ms	10.6MB
10	10^{10}	3 h	10TB
12	10^{12}	13 days	1PB
16	10^{16}	350 years	10EB

figures from Russel and Norvig (2010) for a branching factor 10, assuming 1 million nodes per second can be generated and that a node requires 1000b storage

1 Terrabyte (TB)
= 1024 GB
1 Petabyte (PB)
= 1024 TB = 1048576 GB
1 Exabyte (EB)
= 1024 PB = 1048576 TB
= 1073741824 GB

Age of the universe:
$4.354\neg \pm 0.012 \times 10^{17}$
seconds

Variant: Uniform Cost Search

Step costs may not be equal: two locations adjacent to another in a mesh may be at different distances (represented by a different weight associated to the corresponding edge).

Uniform-cost Search:

- Principle: expand **the node** n **with the lowest path cost** $p(n)$ instead of the shallowest.
- **Priority queue for the frontier**: instead of a FIFO. the INSERT function inserts n while keeping the queue in order with regard to $p(n)$.
- **Goal test**: applied as in general search, when a node is selected for being expanded (not when it is generated).
- **Additional test**: if a "better path" is found

Uniform Cost Search:Pseudo code

```
function UNIFORM-COST-SEARCH(problem)returns solution
    node := a node with State=problem.InitialState, PathCost = 0
    frontier := a priority queue containing only node
    explored :=∅
    loop
        if EMPTY(frontier) then return failure
        node := POP(frontier)
        if problem.GOAL-TEST(node.State) then
            return SOLUTION(node)
        add node.State to explored
        for all action in problem.ACTIONS(node.State) do
            child := CHILD-NODE(problem,node,action)
            if child.State ∉ exploredor frontier then
                frontier := INSERT(CHILD, FRONTIER)
            else
                if child.State ∈ frontier with higher child.Pcost then
                    replace that frontier node with child
```


Some Practice

1. Modify the BFS algorithm by using a priority queue using the path cost function (as in Uniform-cost Search). Exhibit an example evidencing that such an algorithm will not always return the cheapest node.
2. Modify the previous algorithm so that now the Goal test is computed when the node is selected for expansion. Exhibit an example where such an algorithm will not always return the cheapest path.

Some Practice

1. When no Goal test is computed for selecting the node for expansion, the first goal node generated may be on a sub-optimal path.
2. A better path to a node currently on the frontier can be found after a path has been found.
3. What happens if all costs are 0: the algorithm may get stuck in an infinite loop.

Uniform Cost Search Performances

- Completeness: provided all step costs are strictly positive
- Optimality: tick
- Complexity (space and time): can be much greater than b^d

Depth-First Search

Principle:

- Always expands the **deepest** node in the current frontier, eg, proceeding with a child of a last expanded node.
- When a leaf is expanded, it is dropped from the frontier and the search backs up to the next deepest node with unexplored successors.

Implementation, either:

- use a LIFO queue (Last In First Out) with the general search function
- use a recursive function

Depth First Search Performances

- Complete: for graph search in finite state space avoiding repeated states and redundant paths.
- Optimal: ?
- Time complexity: bounded by the size of the state space.
- Space complexity: the good news. Once all descendant have been explored, a node can be removed from memory.

Depth First Search Performances

- Complete: for graph search in finite state space avoiding repeated states and redundant paths.
- Optimal: no
- Time complexity: bounded by the size of the state space.
- Space complexity: the good news. Once all descendant have been explored, a node can be removed from memory.

DFS: Pseudo code, with bounded depth limit

function DEPTH-LIMITED-SEARCH(problem, limit) **returns** solution, failure, cutoff
  return REC-DLS(MAKE-NODE(problem.InitState), problem, limit)

DFS: Pseudo code, with bounded depth limit

```
function REC-DLS(node, problem, limit) returns solution, failure, cutoff
  if problem.GOAL-TEST(node.State) then return SOLUTION(node)
  else
    if limit==0 then return cutoff
    else
      cutoffOccured := false
      for all action in problem.ACTIONS(node.State) do
        child:= CHILD-NODE(problem,node,action)
        result:= REC-DLS(child, problem,limit -1)
        if result=cutoff then
          cutoffOccured := true
        else
          if result != failure then return result
        if cutoffOccured then
          return cutoff
        else
          return failure
```


Variants

- **fixing a depth limit**:
  - especially useful in infinite state spaces: no infinite path anymore
  - new source of incompleteness: if the shallowest node's depth is bigger than the limit
  - non optimal: when limit bigger than the shallowest node's depth
  - time complexity: $\mathcal{O}(b^{limit})$ and space complexity $\mathcal{O}(b * limit)$ (with b branching factor).
- **backtracking search**:
  - only one successor is generated at a time rather than all the successors.
  - possibility to modify instead of creating a new state.
  - allow memory requirements to remain in $\mathcal{O}(m)$ (m maximum depth)

Iterative Deepening Depth First Search

Combines the benefits of depth-first and breadth-first tree search, by successively applying a depth-limited-search to the problem, with increasing limits until a solution is found.
The repetition of the search actually does not impact the complexity too much. This is usually the approach to adopt when the search space is large and the depth of the solution unknown.

```
function ID-S(node, problem, limit) returns solution, failure
  for all depth=0 to ∞ do result:= Depth-Limited-Search(problem, depth)
    if result != cutoff then return result
```

Path Planning

Pathfinding using Heuristic Search

Heuristic search

A Heuristic search is an **Informed search strategy**: uses additional problem-specific knowledge.
Such strategies can find solutions more efficiently than previously described informed search strategies.
We will now consider that the cost is a distance.
Best-first search approaches:

- based on the same general search algorithm
- choice of next node n to expand by minimising value of an evaluation function $f(n)$ (cost estimate).
- implementation is the same than Uniform Cost search
- a **heuristic function** $h(n)$ is used as a component of f providing a *best guess* relying on the problem's specifics.

Heuristic and Evaluation Function for A*

Considering S the starting node and G the goal node, we define the following evaluation function applied to a node:

$$f(N) = g(S \rightsquigarrow N) + h(N \rightsquigarrow G)$$

- g estimates the minimum cost from the starting node to N;
- h is the heuristic estimate of the cost from N to the goal;

$\rightarrow$ f estimates the minimal cost of the path from the start to the goal

What is the * about

Let us define:

- $g*(S \rightsquigarrow N)$ the exact cost of the cheapest path from S to N;
- $h*(N \rightsquigarrow G)$ the exact cost of the cheapest path from N to G;

$\rightarrow$ $f*$ is the exact cost of the optimal path from S to G which goes through N.
When such an $h*$ function is impossible to define, we can sometimes define h as a form of *best guess* using problem-specific features.
For instance: in a 2 dimensional grid where nodes have spatial coordinates, h could be defined as the **euclidean distance** between the nodes.

Computing the actual cost function

$$f(N) = g(S \rightsquigarrow N) + h(N \rightsquigarrow G)$$

- $g(S \rightsquigarrow N)$: actual cost from S to N along the cheapest path *found so far.*
  $\rightarrow$ the value of g may be adjusted downwards if a cheapest path is found during exploration.
- $h(N \rightsquigarrow G)$: knowledge from *outside* the graph: manhattan metric, euclidean distance...

Properties of Heuristic Search - Admissibility

$$f(N) = g(S \rightsquigarrow N) + h(N \rightsquigarrow G)$$

Definition

A search heuristic h is **admissible** (or optimistic) if it **never overestimates** the cost of getting to the goal.
e.g: $\forall N \in problem\, h(N \rightsquigarrow G) \leq h*(N \rightsquigarrow G)$

if h is admissible, f never overestimates the cost of getting from S to G through N.

Properties of Heuristic Search - Consistency

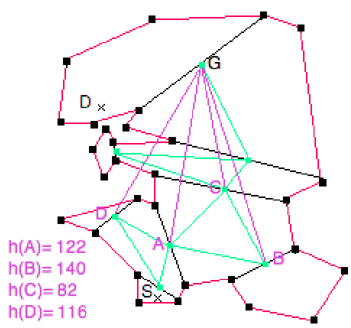
Definition

A search heuristic h is **consistent** (or monotonic, or locally admissible) if for every node N and every successor N' generated by any action a the estimated cost of reaching the goal from N is no greater than the step cost of getting from N to N' plus the estimated cost of reaching the goal from N'
e.g. $\forall N \in problem\, h(N \rightsquigarrow G) \leq c(N, a, N') + h(N' \rightsquigarrow G)$

Exercise: demonstrate that a consistent heuristic is admissible.

Properties of Heuristic Search - Illustration:

Straight line Distance/ euclidean distance heuristic (SLD)



This heuristic is both **admissible** and **consistent**.

Note the heuristic values for respectively A and D. Which node will be expanded first from S?

A* Graph Search

Definition

A* is the algorithm obtained using a node cost function $f(N) = g(S \rightsquigarrow N) + h(N \rightsquigarrow G)$ with the uniform cost search technique, where g is the actual cost of getting to the node and h a heuristic function estimating the cost of getting from the node to the goal.

Theorem

Graph search using A is optimal if the heuristic function is consistent.*

A* Graph Search Optimality

We can show that:

- if $h(N)$ is consistent then the values of $f(N)$ along any path will not be decreasing.
- whenever A* selects a node for expansion, the optimal path to that node has been found.

The sequence of nodes expanded by A* for graph search is thus non decreasing for f(N).
(do it now)

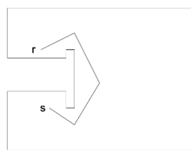
A* Complexity and Variants

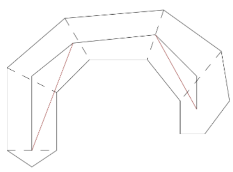
In the worse case (of heuristic), A* complexity is exponential both in memory and time.
A couple of variants (many more):

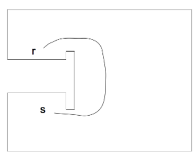
- Iterative Deepening A*: Using the same principles as ID-DFS, the search is applied within a frontier defining the maximum value of heuristic function and extending it if the search did not succeed (spares the cost of maintaining an open list)
- Real-Time A* (with a learning version): interleaves moves to the next position on best path found so far with searching for path, taking the new position as a start of the pathfinding problem.

Last step: animate properly

- smoothing the path
- test line of sight for shortcuts
- avoid dynamic obstacles using repulsion vector







Wrap-up, Conclusion

- AI playing a game $\neq$ AI for a *fun* game
- Synthetic characters for a game world = significantly easier than in the real world
- pathfinding= Discretisation of the game world + graph search algorithms + animation to smoothe the moves.

Additional Readings

Artificial Intelligence, a Modern Approach. Russel and Norvig. Third Edition, 2010.
Algorithms and Networking for Computer Games. Smed and Hakonen. 2006
Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Shaker, Togelius and Nelson. 2016.