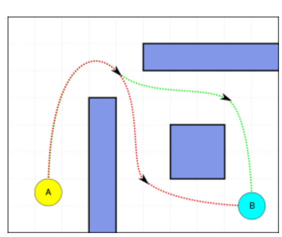
3D Computer Game Programming

Navigation & Pathfinding

Introduction

- 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



Pathfinding

- Two key aspects of pathfinding:
  - Representing the search space
  - Searching for a path

Representing the Search Space

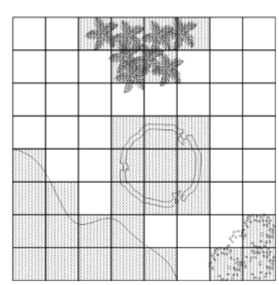
- Agents need to know where they can move
- Search space should represent either
  - Clear routes that can be traversed
  - Or the entire walkable surface
- Search space typically doesn't represent:
  - Small obstacles or moving objects

Representing the Search Space

- Most common search space representations are
  - **Grids**
  - **Waypoint graphs**
  - **Navigation meshes**

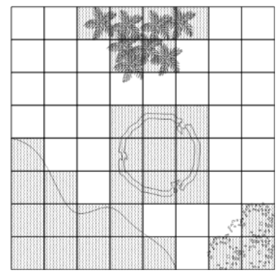
Grids

- 2D grids – intuitive world representation
  - Works well for many games including some 3D games such as *Warcraft III*
- Each cell is flagged
  - *Passable* or *impassable*
- Each object in the world can occupy one or more cells



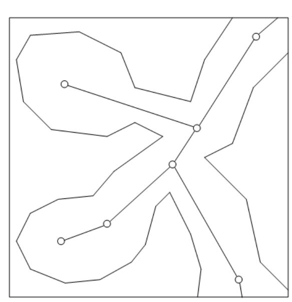
Characteristics of Grids

- Fast look-up
- Easy access to neighboring cells
- Complete representation of the level



Waypoint Graph

- A waypoint graph specifies lines/routes that are “safe” for traversing
- Each line (or link) connects exactly two waypoints

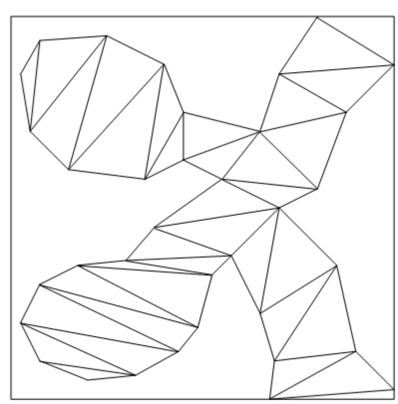


Characteristics of Waypoint Graphs

- Waypoint node can be connected to any number of other waypoint nodes
- Waypoint graph can easily represent arbitrary 3D levels
- Can incorporate auxiliary information
  - Such as ladders and jump pads
- **Incomplete** representation of the level

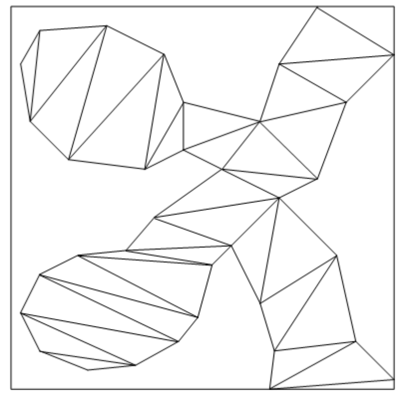
Navigation Meshes

- Combination of grids and waypoint graphs
- Every node of a navigation mesh represents a **convex** polygon (or area)
  - As opposed to a single position in a waypoint node



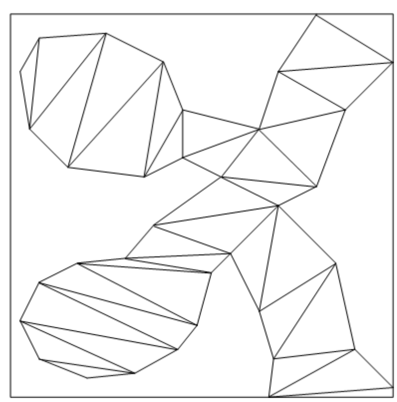
Navigation Meshes (continued)

- Advantage of **convex** polygon
  - Any two points inside can be connected without crossing an edge of the polygon
- Navigation mesh can be thought of as a walkable surface.



Characteristics of Navigation Meshes

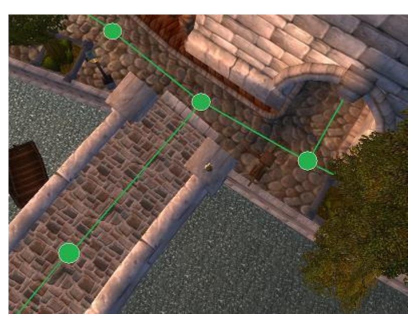
- **Complete** representation of the level
- Ties pathfinding and collision detection together
- Can easily be used for 2D and 3D games.
  - Unlike a 2D grid it allows traversable areas that overlap above and below at different heights.



Waypoint Graph vs Navigation Mesh

Waypoint Graph

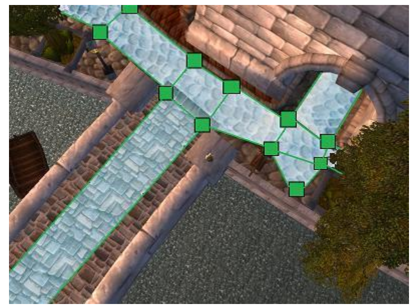
Part of Stormwind City in World of WarCraft



The same area annotated with a waypoint graph

Navigation Mesh

Part of Stormwind City in World of WarCraft



The same area annotated with a navigation mesh

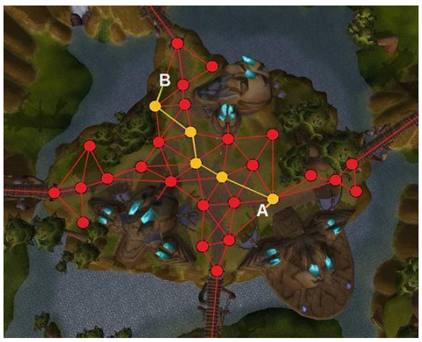
Waypoint Graph VS Navigation Mesh (1)

The town of Halaa in World of WarCraft, seen from above

- With a waypoint-based system, we need to place a lot of waypoints to get full coverage producing adequate movements.

Waypoint Graph VS Navigation Mesh (2)

- A waypoint-based system makes characters "zig-zag" as they move.

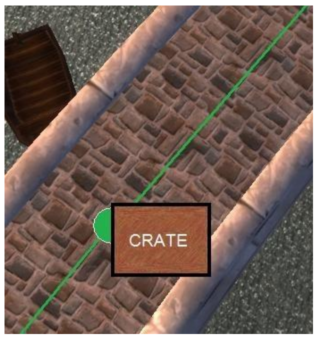


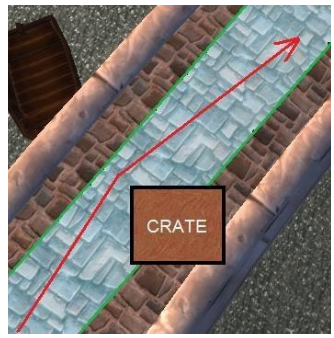
Waypoint Graph VS Navigation Mesh (2) – cont’d

- A waypoint path (red) and a smoothed navigation mesh path (blue)

Waypoint Graph VS Navigation Mesh (3)

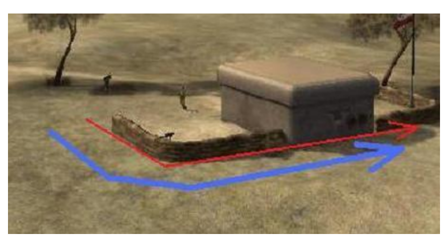
- Path correction - A waypoint-based system makes robust dynamic obstacle avoidance difficult, if not impossible.





Waypoint Graph VS Navigation Mesh (4)

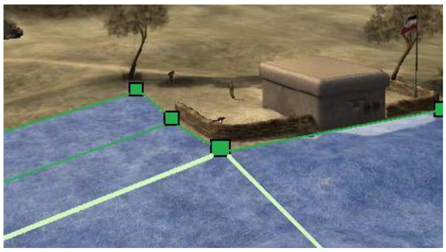
A concrete bunker



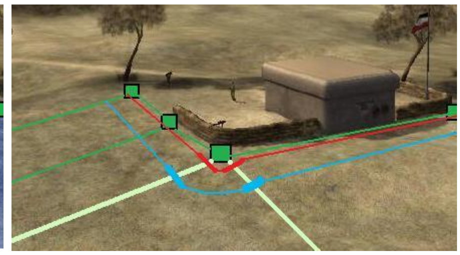
The closest possible paths that a human soldier (red) and a tank (blue) can take around the sandbags

- A waypoint-based system does not work robustly for characters that move differently.

Waypoint Graph VS Navigation Mesh (4) – cont'd



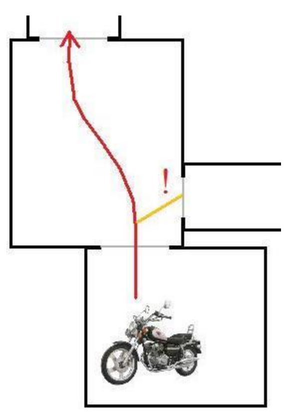
Part of a navigation mesh around the outside of the bunker



Part of a navigation mesh around the outside of the bunker

- You need to use totally separate waypoint networks for the tanks and the soldiers.

Waypoint Graph VS Navigation Mesh (4) – cont'd



A motorcyclist can go into the room at the top (red) but not the room on the right (orange)

- With a navigation mesh
  - Test the turn angles and distances as doing the pathfinding and reject any paths that require steep turns over short distances.
- With a waypoint approach
  - At a minimum, we'd need a completely separate waypoint graph for motorcycles, since we can't use the same graph that a person are using.

Search (Pathfinding) Algorithms

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
  - Grids
  - Waypoint graphs
  - Navigation meshes (pathfinding between polygons in the mesh)

Pathfinding Algorithms

- Random walk
- Random trace
- Search Algorithms
  - Breadth first search
  - Best first search (Greedy search)
  - A* algorithm
  - Dijkstra’s algorithm

Criteria for Evaluating Pathfinding Algorithms

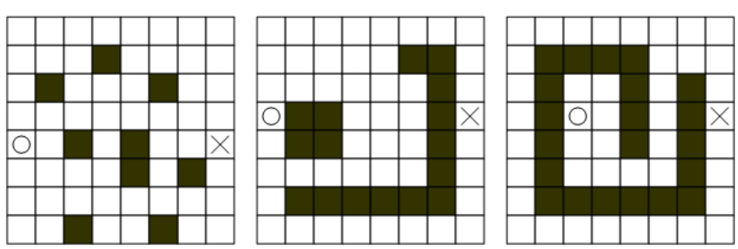
- Is it a complete algorithm
  - A complete algorithm guarantees to find a path if one exists.
- Is it optimal algorithm?
  - a Quality of final path (is it shortest?)
- Complexity -Resource consumption during search
  - Time (CPU) and Space (memory)

Random Walk

- Algorithm
  - Takes a random step
  - If goal reached, then done
  - Repeat procedure until goal reached
- Add intelligence
  - Only step in cases where distance to goal is smaller
  - Stuck? With a certain probability, allow a step where distance to goal is greater

Random Walk (continued)

- How will Random Walk do on the following maps?



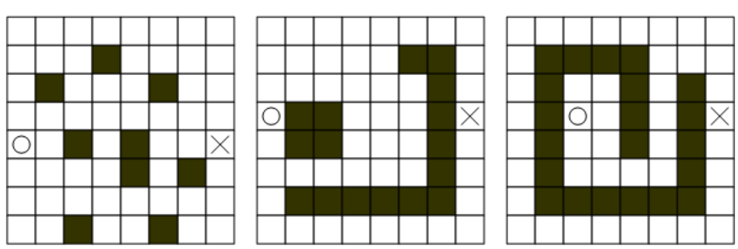
Going from o to x (or vice versa)

Random Trace

- 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?

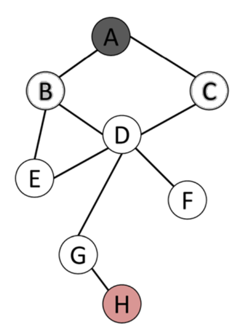


Random Walk and Trace Characteristics

- Not ***complete*** algorithms
- Found paths are unlikely to be optimal
- Consumes very little memory

Graph Search Algorithms for Pathfinding

- Breadth First Search (BFS)
- Best first search (Greedy search)
- A* algorithm
- Dijkstra's algorithm

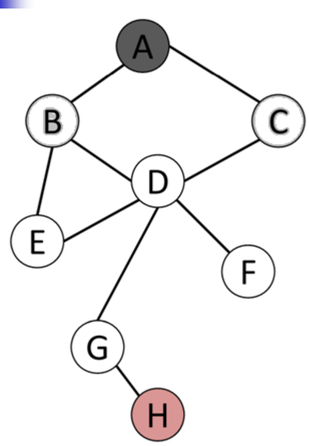


- Work on a graph with nodes.
  - All search spaces can be represented as a graph.
- A list of nodes represents a path.

Graph Search Algorithms

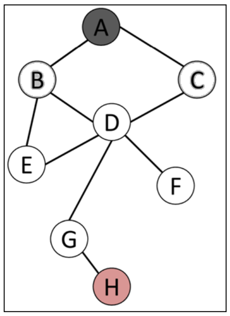
- Maintain two lists:
  - **Closed List** storing visited nodes
  - **Open List** storing nodes to be visited
- Algorithm:
  - Select a node in the open list for expansion
  - Check to see if it is the goal.
  - If it is, the path is found. Return.
  - If not,
    - Expand it - check its edges and add nodes to the open lists
    - Place current node on closed list (mark a field as visited)
  - Repeat

Breadth-First Search (BFS)

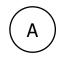


- Start node : A
- Destination node: H

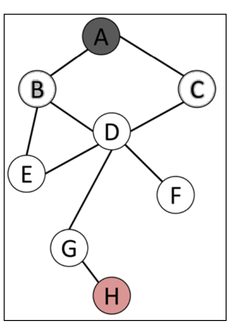
Breadth-First Search (BFS)



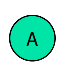
- Initially,
  - Closed List = { }
  - Open List = {A}



Breadth-First Search (BFS)

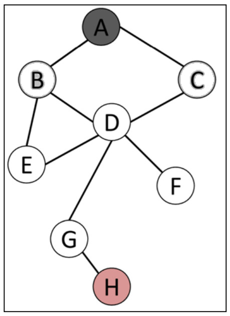


- Initially,
  - Closed List = { }
  - Open List = {A}

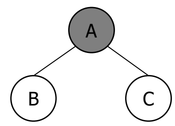


Is this goal?

Breadth-First Search (BFS)

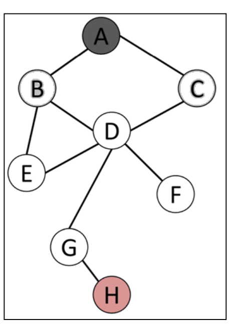


- Closed List = {A}
- Open List = {B,C}

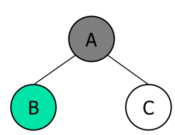


Expand the node and move it to the closed list.

Breadth-First Search (BFS)

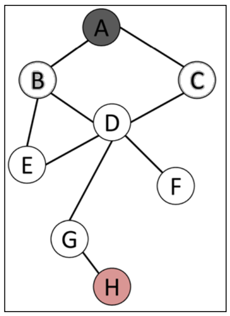


- Closed List = {A}
- Open List = {B,C}

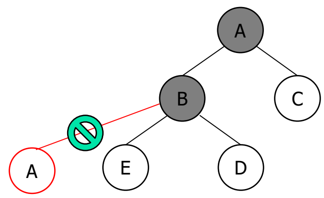


Is this goal?

Breadth-First Search (BFS)

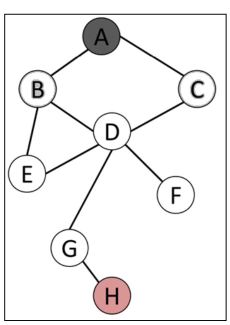


- Closed List = {A,B}
- Open List = {C, E, D}

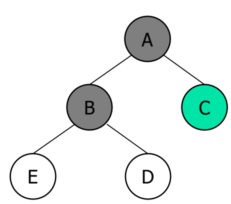


Add unvisited nodes (i.e. exclude nodes in the closed list)

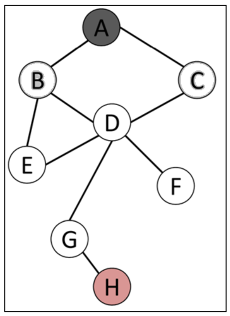
Breadth-First Search (BFS)



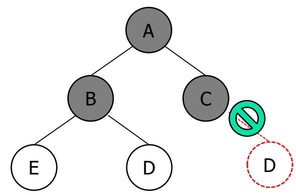
- Closed List = {A,B}
- Open List = {C, E, D}



Breadth-First Search (BFS)

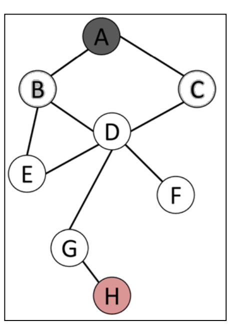


- Closed List = {A,B,C}
- Open List = {E, D}

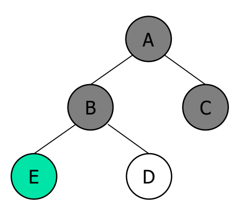


Add unexplored nodes (i.e. exclude any node in closed or open lists)

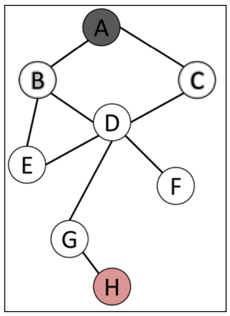
Breadth-First Search (BFS)



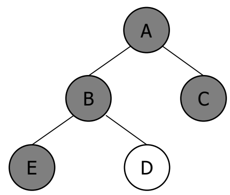
- Closed List = {A,B,C}
- Open List = {E, D}



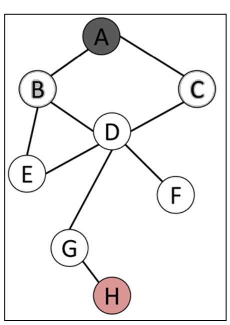
Breadth-First Search (BFS)



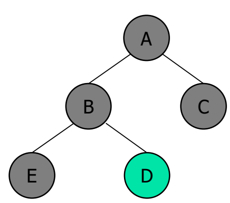
- Closed List = {A,B,C,E}
- Open List = {D}



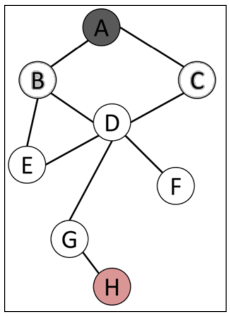
Breadth-First Search (BFS)



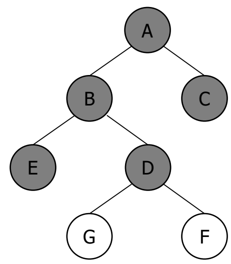
- Closed List = {A,B,C,E}
- Open List = {D}



Breadth-First Search (BFS)

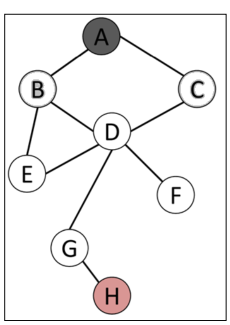


- Closed List = {A,B,C,E,D}
- Open List = {G, F}

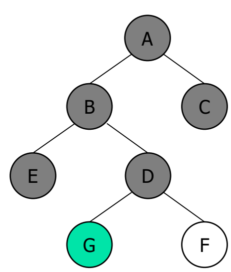


Exclude visited node for expansion.

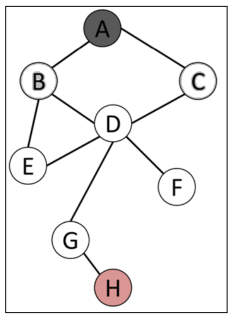
Breadth-First Search (BFS)



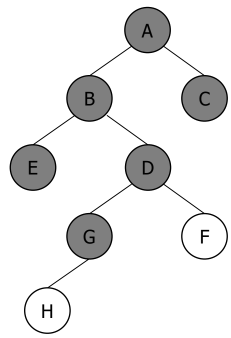
- Closed List = {A,B,C,E,D}
- Open List = {G, F}



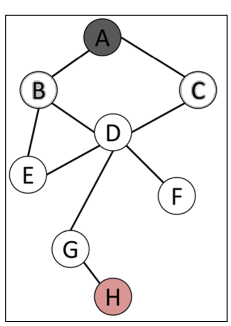
Breadth-First Search (BFS)



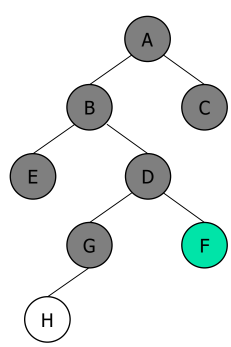
- Closed List = {A,B,C,E,D,G}
- Open List = {F,H}



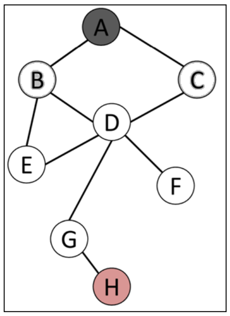
Breadth-First Search (BFS)



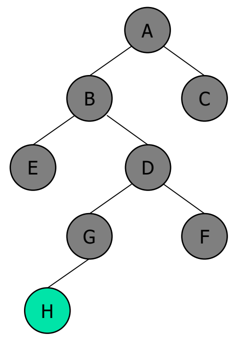
- Closed List = {A,B,C,E,D,G}
- Open List = {F,H}



Breadth-First Search (BFS)



- Closed List = {A,B,C,E,D,G,F}
- Open List = {H}



Found the path: A->B->D->G->H

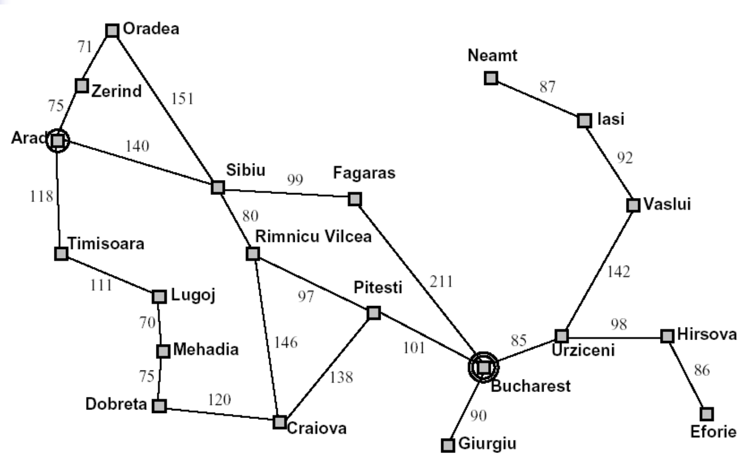
Properties of BFS

- Complete- Yes (if *the space* is finite)
- Optimal- Yes if edge/step costs are same; not optimal in general
- Time and Space complexity is high.

Node Selection

- Which open node becomes closed first?
  - **Breadth-First** processes the node that has been waiting the longest => implements open list using a queue.
  - **Best-First (Greedy)** processes the one that is closest to the goal
  - **A*** chooses a node that is cheap and close to the goal
  - **Dijkstra's** processes the one that is the cheapest according to some weight

Simplified road map of part of Romania



Best-First Search (Greedy Search)

- Idea: select a node based on an evaluation function, *f(n),* for expansion
  - f(n) : estimate of “desirability“
  - Expand most desirable unexpanded/unvisited node in the Open List
- Implementation
  - Open List is a queue sorted in decreasing order of desirability (usually, in increasing order of *f*)

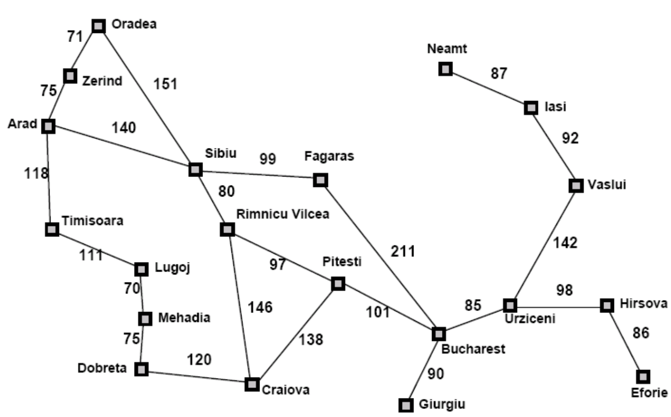
Heuristic Function

- Heuristic function *h(n)*
  - *h(n)* = <u>estimated</u> cost of the cheapest path from node *n* to a goal node
    - e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
  - $h(n) \geq 0$, so $h(G) = 0$ for any goal *G*.

Greedy Search

- Evaluation function *f(n)* = *h(n)* (heuristic)
  - estimate of cost from *n* to the closest goal
  - Eg. $f(n) = h_{SLD}(n)$
- Greedy search expands the node that **appears** to be closest to goal at each step.

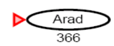
Romania w/ Step Cost in km



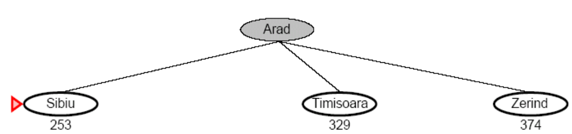
Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

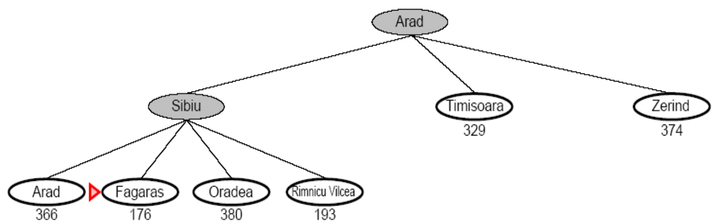
Greedy Search Example



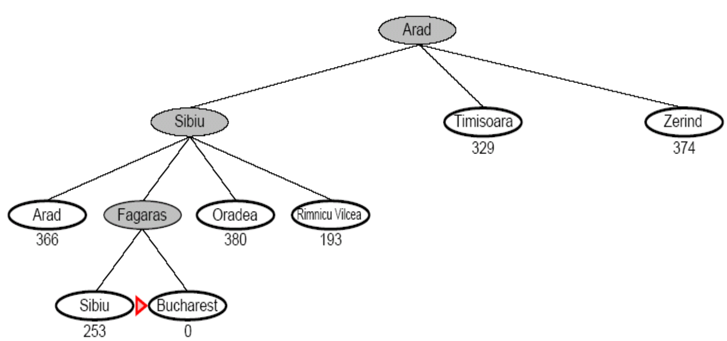
Greedy Search Example



Greedy Search Example



Greedy Search Example



3D Computer Game Programming

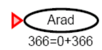
Properties of Greedy Search

- Complete??
  - No - can get stuck in loops,
    - e.g., with Oradea as goal, Iasi → Neamt → Iasi → Neamt →
  - Complete in finite space with repeated-state checking
- The algorithm follows a single path all the way and backup when it hits a dead end. But a good heuristic can give dramatic improvement
- Space?? - keeps all nodes in memory
- Optimal?? No

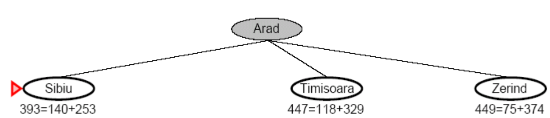
A* Search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = path cost from the start node to reach n
  - $h(n)$ = *estimated* cost from n to goal
  - $f(n)$ = *estimated* total cost of path through n to goal
- Heuristic function is admissible if $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n.
  - E.g., $h_{SLD}(n)$ is admissible because it never overestimates the actual road distance
- A* search uses an **admissible** heuristic.

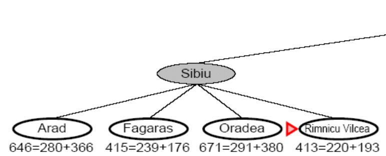
A* Search



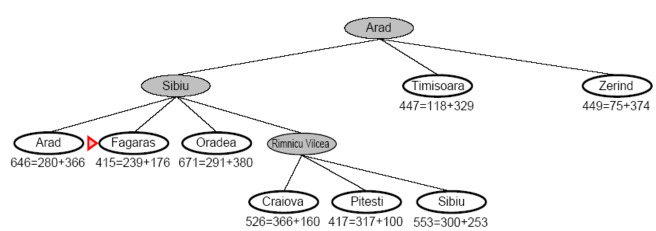
A* Search



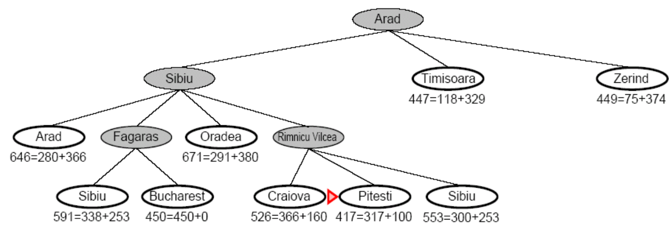
A* Search



A* Search

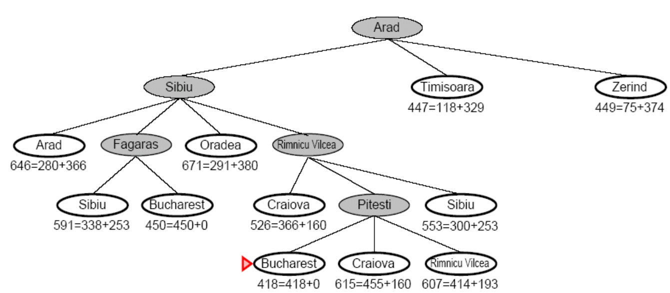


A* Search



3D Computer Game Programming

A* Search

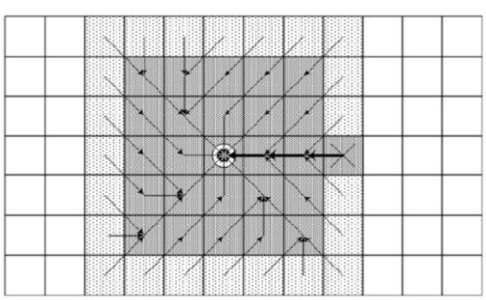


3D Computer Game Programming

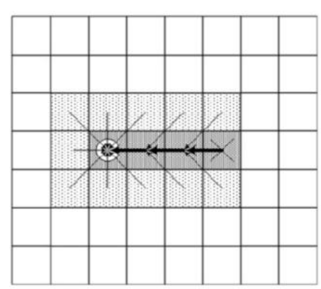
Properties of A*

- Complete?? Yes,
- Theorem: A* search is optimal (if *h(n)* is admissible)

Comparison 1

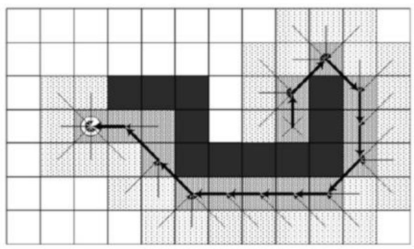


Breath-First Search

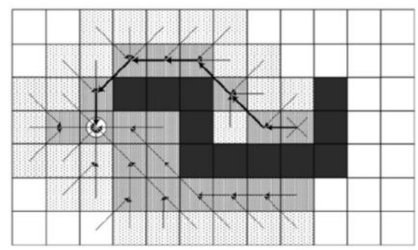


Best-First Search
(Greedy Search)

Comparison 2



Best-First Search
(Greedy Search)

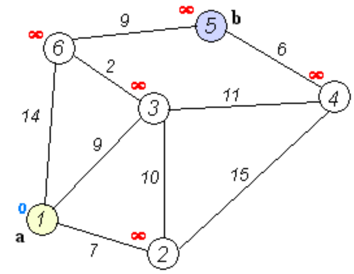


A* Search

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

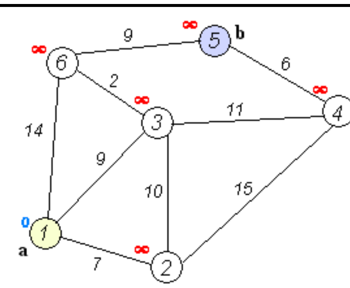


Shortest path from a **source** to all targets.

Data structures:

1. `dist[v]`: stores distance from source to any node v
2. `previous[v]`: previous node in optimal path from source

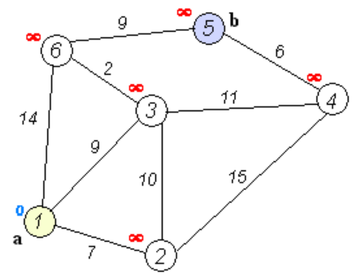
Dijkstra



Shortest path from a source to all targets.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:           // Initializations
        dist[v] := infinity               // Unknown distance function from source to v
        previous[v] := undefined          // Previous node in optimal path from source
    dist[source] := 0                     // Distance from source to source
    Q := the set of all nodes in Graph
    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                 // The main loop
        u := vertex in Q with smallest dist[]
        if dist[u] = infinity:
            break                         // all remaining vertices are inaccessible
        remove u from Q
        for each neighbor v of u:         // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]:             // Relax (u,v,a)
                dist[v] := alt
                previous[v] := u
    return previous[]
```

Dijkstra



Getting a path from a source to a target by tracing previous[] backward.

```
S := empty sequence
u := target
while previous[u] is defined:
    insert u at the beginning of S
    u := previous[u]
```


Dijkstra Characteristics

- Exhaustive search
- At least as resource intensive as Breadth-First
- Always finds the most optimal path
- Complete algorithm

Summary

- Reviewed search space representations used in Game Programming.
- NavMesh is commonly used.

- Reviewed pathfinding algorithms used in Game Programming.
- A* and its variants are commonly used in Games.