Game Programming

Bing-Yu Chen
National Taiwan University

Game AI

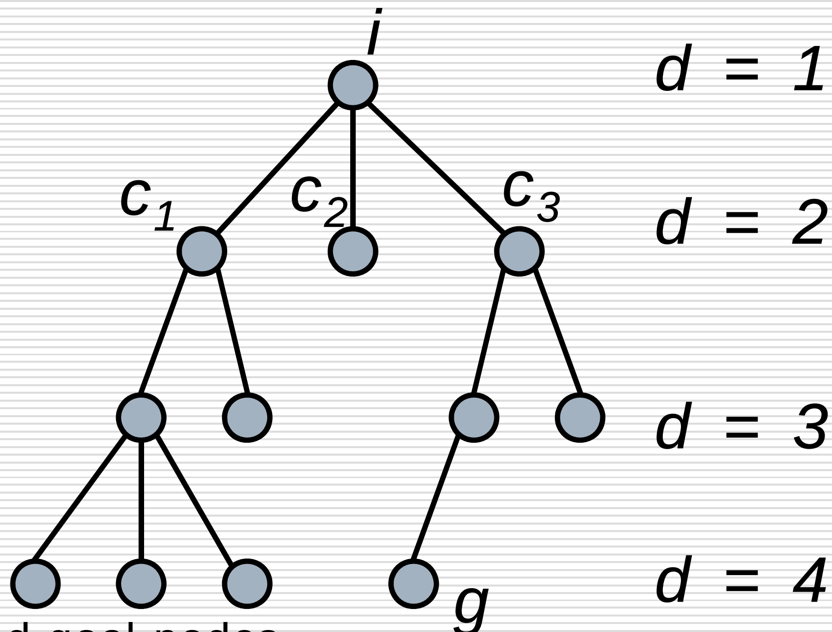
- Search
- Path Finding
- Finite State Machines
- Steering Behavior

Search

- Blind search
  - Breadth-First Search
  - Depth-First Search
- Heuristic search
  - A*
- Adversary search
  - MinMax

Introduction to Search

- Using tree diagram (usually) to describe a search problem
  - Search starts
    - Node *i*
  - Goal
    - Goal node *g*
  - Successors
    - c_1, c_2, c_3...
  - Depth
    - d = 0, 1, 2, ...



- Search problem
  - Input
    - Description of the initial and goal nodes
    - A procedure that produces the successors of an arbitrary node
  - Output
    - A legal sequence of nodes starting with the initial node and ending with the goal node.

Search Examples in Traditional AI

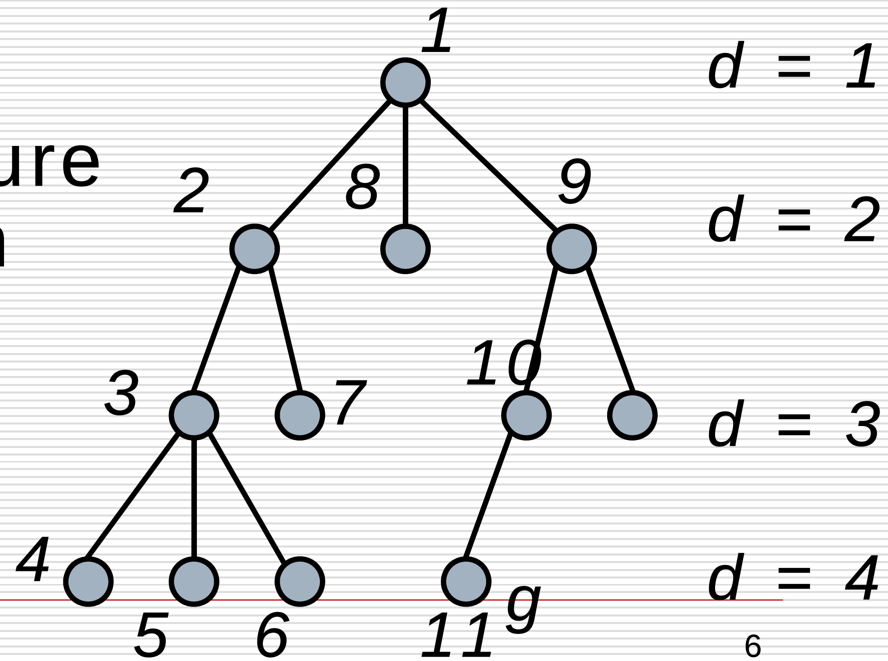
- Game playing
  - Chess
  - Backgammon
- Finding a path to goal
  - The towers of Hanoi
  - Sliding tile puzzles
    - 8 puzzles
- Simply finding a goal
  - n-queens

Search Algorithm

1. Set ***L*** to be a list of the initial nodes. At any given point in time, ***L*** is a list of nodes that have not yet been examined.
2. If ***L*** is empty, failed. Otherwise, pick a node ***n*** from ***L***.
3. If ***n*** is the goal node, stop and return it and the path from the initial node to ***n***.
4. Otherwise, remove ***n*** from ***L*** and add to ***L*** all of ***n***'s children, labeling each with its path from the initial node.
5. Return to Step 2.

Depth-First Search

- Always exploring the child of the most recently expanded node
- Terminal nodes being examined from left to right
- If the node has no children, the procedure backs up a minimum amount before choosing another node to examine.



Depth-First Search

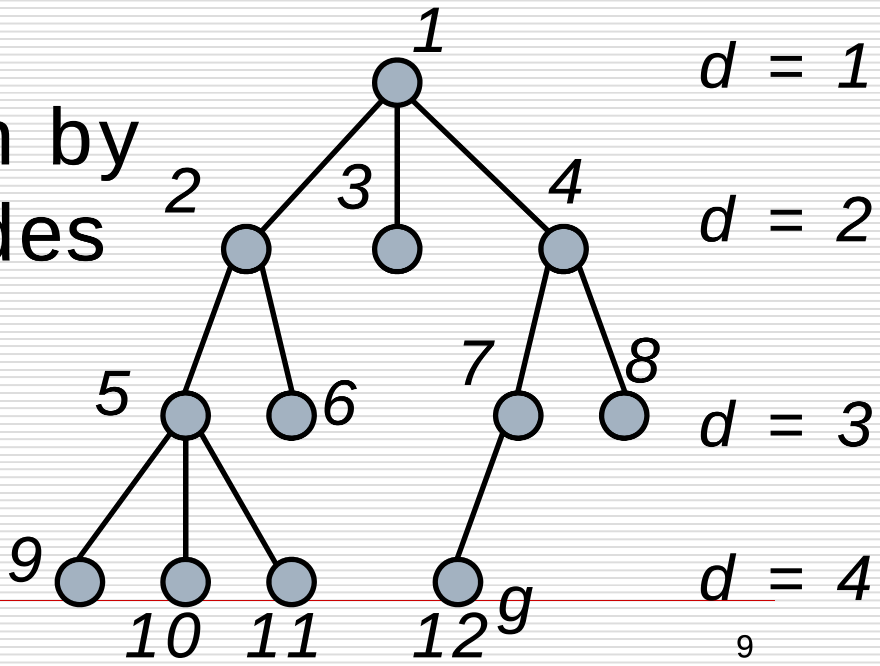
- We stop the search when we select the goal node ***g***.
- Depth-first search can be implemented by pushing the children of a given node onto the front of the list ***L*** in Step 4. of Search Algorithm.
- And always choosing the first node on ***L*** as the one to expand.

Depth-First Search Algorithm

1. Set ***L*** to be a list of the initial nodes.
2. If ***L*** is empty, failed. Otherwise, pick a node ***n*** from ***L***.
3. If ***n*** is the goal node, stop and return it and the path from the initial node to ***n***.
4. Otherwise, remove ***n*** from ***L*** and add to *the* ***front*** *of* ***L*** all of ***n***'s children, labeling each with its path from the initial node.
5. Return to Step 2.

Breadth-First Search

- The tree examined from top to down, so every node at depth ***d*** is examined before any node at depth ***d* + *1***.
- We can implement breadth-first search by adding the new nodes to the end of the list ***L***.

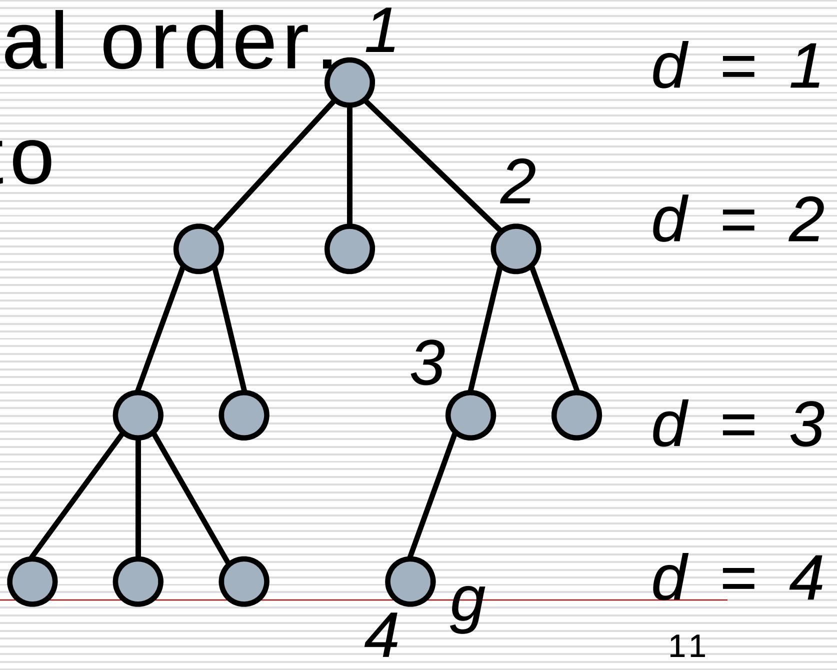


Breadth-First Search Algorithm

1. Set ***L*** to be a list of the initial nodes.
2. If ***L*** is empty, failed. Otherwise, pick a node ***n*** from ***L***.
3. If ***n*** is the goal node, stop and return it and the path from the initial node to ***n***.
4. Otherwise, remove ***n*** from ***L*** and add to *the **end** of **L*** all of ***n***'s children, labeling each with its path from the initial node.
5. Return to Step 2.

Heuristic Search

- Neither depth-first nor breadth-first search
- Exploring the tree in anything resembling an optimal order.
- Minimizing the cost to solve the problem.



Heuristic Search

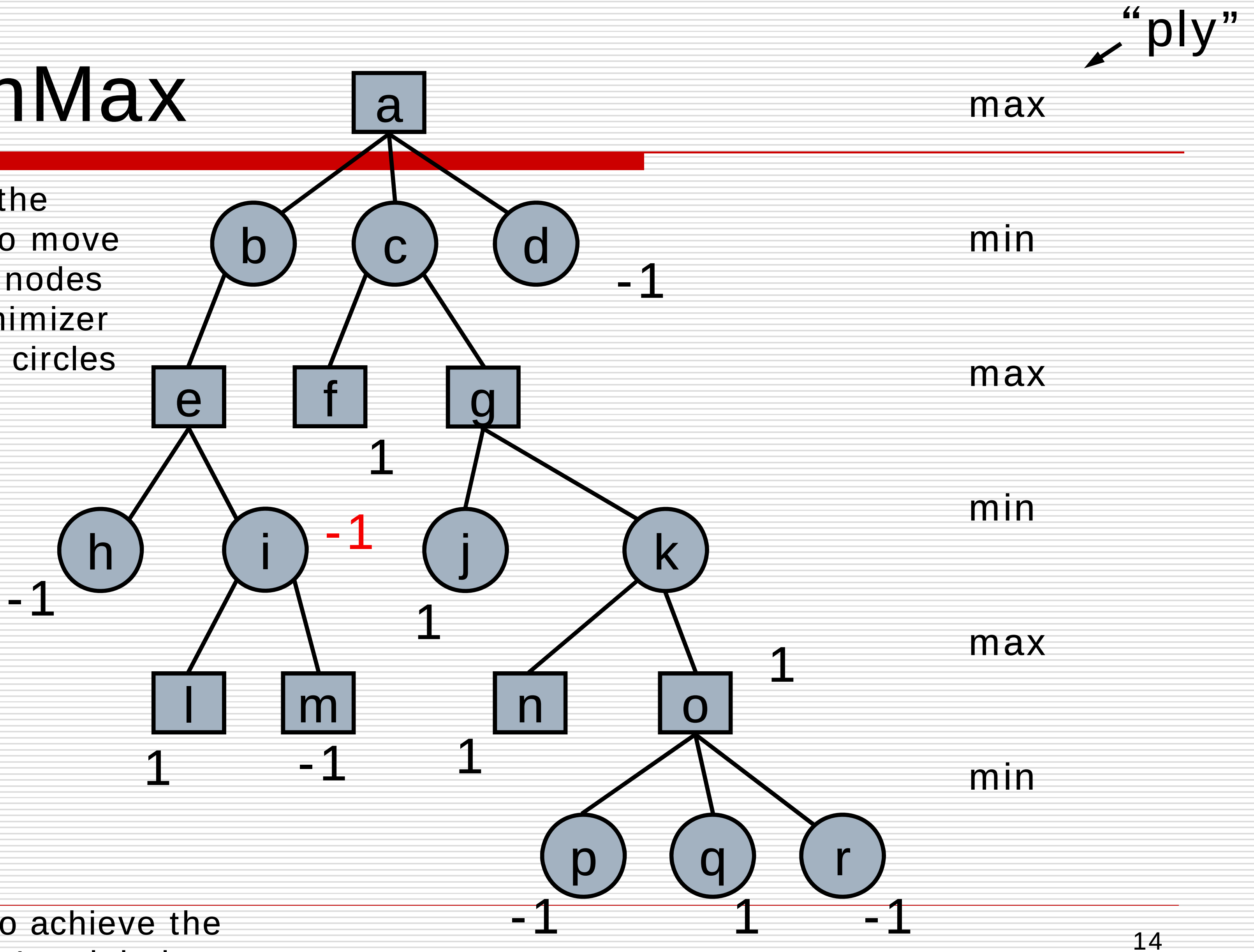
- When we picking a node from the list ***L*** in Step 2. of Search Algorithm, what we will do is to remove steadily from the root node toward the goal by always selecting a node that is as close to the goal as possible.
  - Estimated by distance and minimizing the cost?
    - A* !

Adversary Search

- Assumptions
  - Two-person games in which the players alternate moves.
  - They are games of “perfect” information, where the knowledge available to each player is the same.
- Examples :
  - Tic-tac-toe
  - Checkers
  - Chess
  - Go
  - Othello
  - Backgammon
- Imperfect information
  - Pokers
  - Bridge

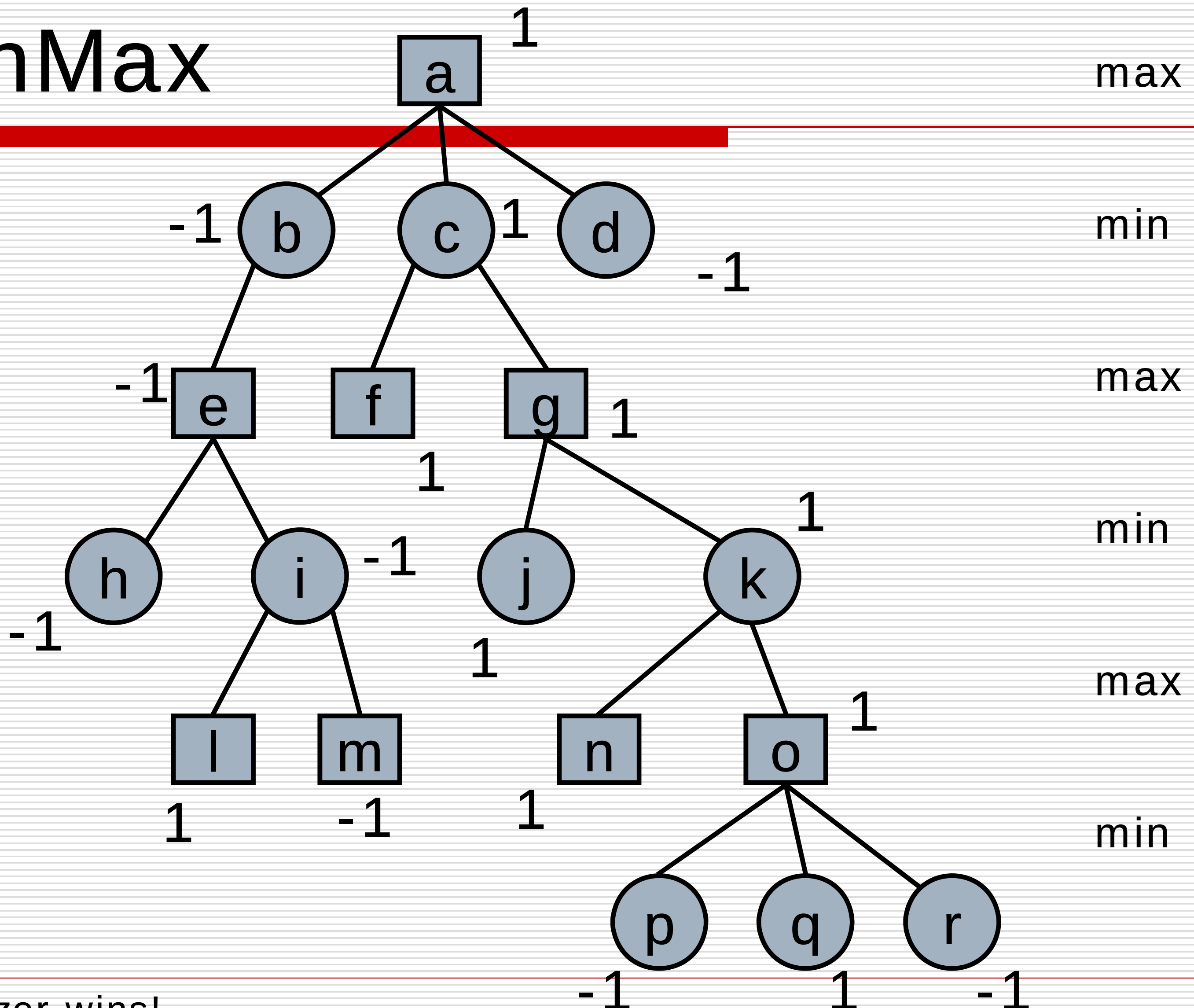
MinMax

Nodes with the maximizer to move are square; nodes with the minimizer to move are circles



Maximizer to achieve the Outcome of 1; minimizer to Achieve the outcome of -1

MinMax



The maximizer wins!

MinMax Idea

1. Expand the entire tree below ***n***.
2. Evaluate the terminal nodes as wins for the minimizer or maximizer.
3. Select an unlabelled node all of whose children have been assigned values. If there is no such node, return the value assigned to the node ***n***.
4. If the selected node is one at which the minimizer moves, assign it a value that is the minimum of the values of its children. If it is a maximizing node, assign it a value that is the maximum of the children's values. Return to Step 3.

MinMax Algorithm

1. Set $\boldsymbol{L} = \{ \boldsymbol{n} \}$, the unexpanded nodes in the tree.
2. Let $\boldsymbol{x}$ be the 1st node on $\boldsymbol{L}$. If $\boldsymbol{x} = \boldsymbol{n}$ and there is a value assigned to it, return this value.
3. If $\boldsymbol{x}$ has been assigned a value $\boldsymbol{v_x}$, let $\boldsymbol{p}$ be the parent of $\boldsymbol{x}$ and $\boldsymbol{v_p}$ the value currently assigned to $\boldsymbol{p}$. If $\boldsymbol{p}$ is a minimizing node, set $\boldsymbol{v_p} = \min(\boldsymbol{v_p}, \boldsymbol{v_x})$. If $\boldsymbol{p}$ is a maximizing node, set $\boldsymbol{v_p} = \max(\boldsymbol{v_p}, \boldsymbol{v_x})$. Remove $\boldsymbol{x}$ from $\boldsymbol{L}$ and return to Step 2.
4. If $\boldsymbol{x}$ has not been assigned a value and is a terminal node, assign it the value **1** or **-1** depending on whether it is a win for the maximizer or minimizer respectively. Assign $\boldsymbol{x}$ the value **0** if the position is a draw. Leave $\boldsymbol{x}$ on $\boldsymbol{L}$ and return to Step 2.
5. Otherwise, set $\boldsymbol{v_x}$ to be $\boldsymbol{-\infty}$ if $\boldsymbol{x}$ is a maximizing node and $\boldsymbol{+\infty}$ if $\boldsymbol{x}$ is a minimizing node. Add the children of $\boldsymbol{x}$ to the front of $\boldsymbol{L}$ and return to Step 2.

MinMax

- Some issues
  - Draw
  - Estimated value **e(*n*)**
    - **e(*n*) = 1** : the node is a win for maximizer
    - **e(*n*) = -1** : the node is a win for minimizer
    - **e(*n*) = 0** : that is a draw
    - **e(*n*) = -1 ~ 1** : the others
  - When to decide stop the tree expanding further ?

MinMax Algorithm

1. Set ***L*** = { ***n*** }, the unexpanded nodes in the tree.
2. Let ***x*** be the 1st node on ***L***. If ***x*** = ***n*** and there is a value assigned to it, return this value.
3. If ***x*** has been assigned a value $\boldsymbol{v_x}$, let ***p*** be the parent of ***x*** and $\boldsymbol{v_p}$ the value currently assigned to ***p***. If ***p*** is a minimizing node, set $\boldsymbol{v_p}$ = min($\boldsymbol{v_p}$, $\boldsymbol{v_x}$). If ***p*** is a maximizing node, set $\boldsymbol{v_p}$ = max($\boldsymbol{v_p}$, $\boldsymbol{v_x}$). Remove ***x*** from ***L*** and return to Step 2.
4. If ***x*** has not been assigned a value and is a terminal node, assign it the value **1** or **-1** depending on whether it is a win for the maximizer or minimizer respectively. Assign ***x*** the value **0** if the position is a draw. Leave ***x*** on ***L*** and return to Step 2.
5. Otherwise, set $\boldsymbol{v_x}$ to be **−∞** if ***x*** is a maximizing node and **+∞** if ***x*** is a minimizing node. Add the children of ***x*** to the front of ***L*** and return to Step 2.

MinMax Algorithm (final)

1. Set ***L*** = { ***n*** }, the unexpanded nodes in the tree.
2. Let ***x*** be the 1st node on ***L***. If ***x*** = ***n*** and there is a value assigned to it, return this value.
3. If ***x*** has been assigned a value v_x, let ***p*** be the parent of ***x*** and v_p the value currently assigned to ***p***. If ***p*** is a minimizing node, set $v_p = \min(v_p, v_x)$. If ***p*** is a maximizing node, set $v_p = \max(v_p, v_x)$. Remove ***x*** from ***L*** and return to Step 2.
4. If ***x*** has not been assigned a value and either ***x*** is a terminal node or *we have decided not to expand the tree further, compute its value using the evaluation function*. Leave ***x*** on ***L*** and return to Step 2.
5. Otherwise, set v_x to be $-\infty$ if ***x*** is a maximizing node and $+\infty$ if ***x*** is a minimizing node. Add the children of ***x*** to the front of ***L*** and return to Step 2.

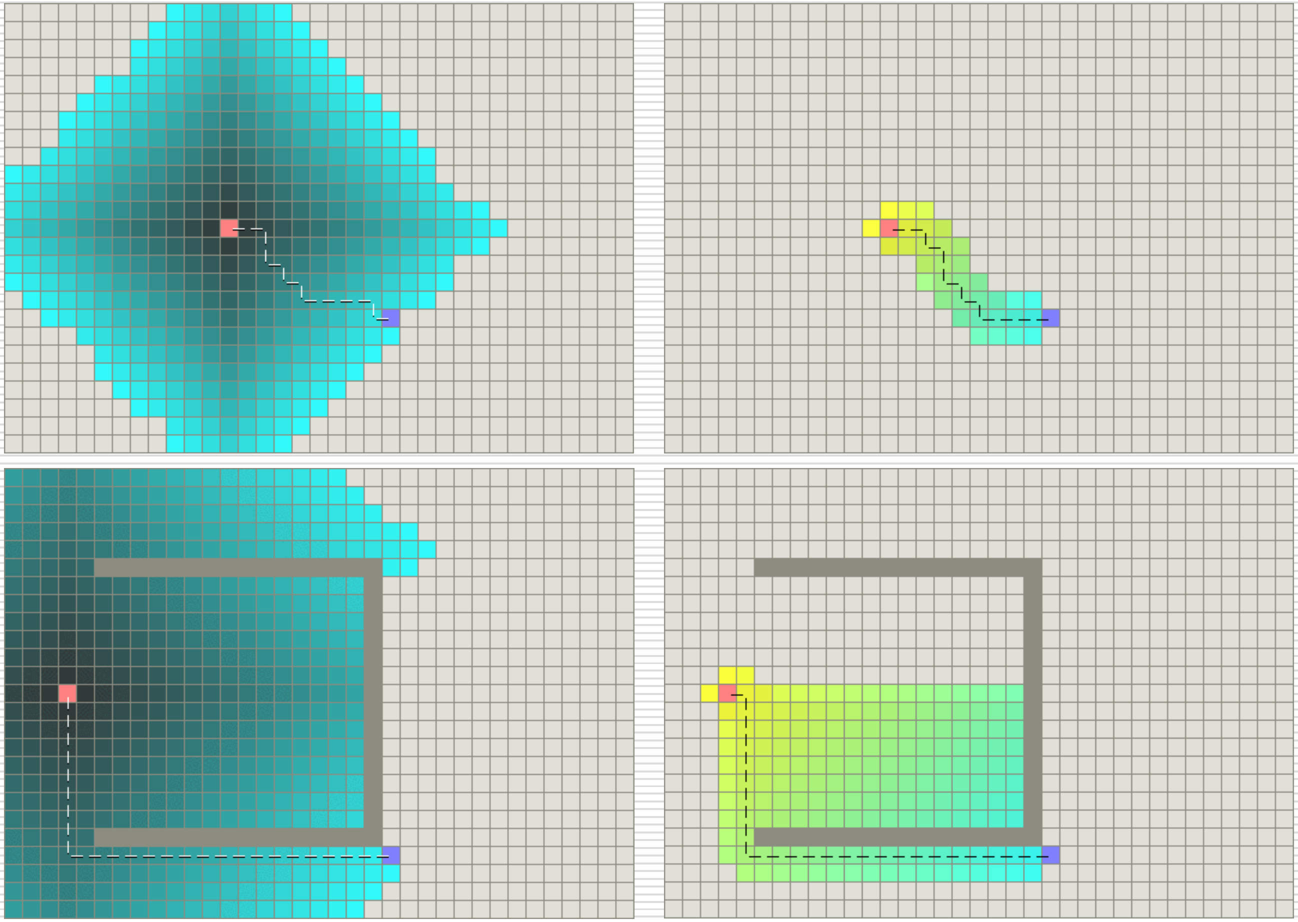
Introduction to Path Finding

- A common situation of game AI
- Path planning
  - From start position to the goal
- Most popular technique
  - A* (A Star)
    - 1968
    - A search algorithm
    - Favorite teaching example : 15-pizzule
    - Algorithm that searches in a state space for the least costly path from start state to a goal state by examining the neighboring states

Dijkstra vs. A*

Without obstacle

With obstacle



Dijkstra A*

Dijkstra vs. A*

- Dijkstra: compute the optimal solution
- Diskstra: search space much larger than A*
- A*: simple
- A*: fast
- A*: “good” result
- A*: employ heuristic estimate to eliminate many paths with high costs -> speedup process to compute satisfactory “shortest” paths

A*: cost functions

- Goal: compute a path from a start point ***S*** to a goal point ***G***
- Cost at point ***n***:
  f(*n*) = g(*n*) + h(*n*)
- **g(*n*)**: distance from the start point ***S*** to the current point ***n***
- **h(*n*)**: estimated distance from the current point ***n*** to the goal point ***G***
- **f(*n*)**: current estimated cost for point ***n***

A*: cost functions

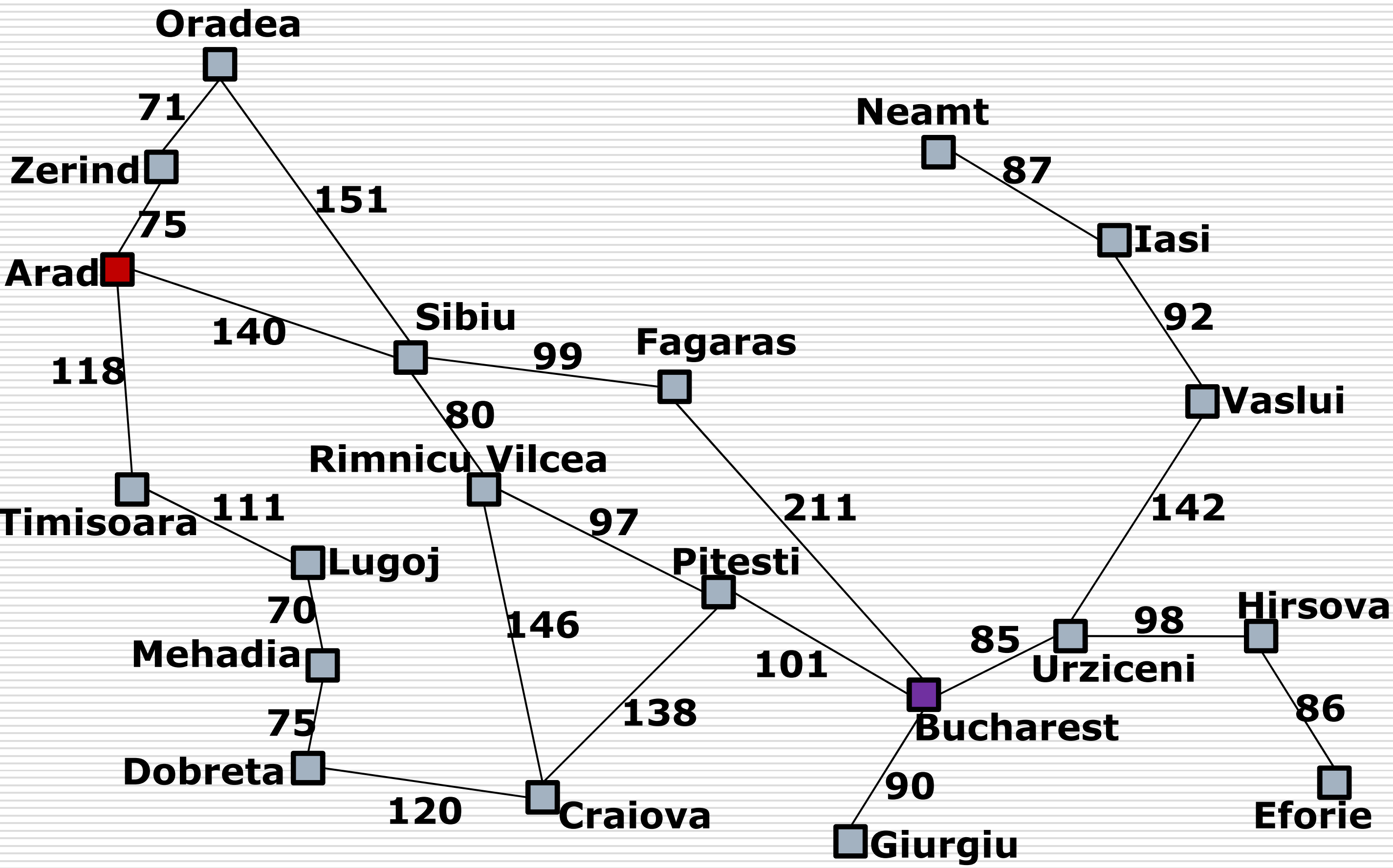
- The role of **h(*n*)**
  - A major cost evaluation function of A*
  - Guide the performance of A*

- **d(*n*)**: the actual distance between ***S*** and ***G***
- **h(*n*) = 0** : A* is equivalent to Dijkstra algorithm
- **h(*n*) <= d (*n*)** : guarantee to compute the shortest path; the lower the value **h(*n*)**, the more node A* expands
- **h(*n*) = d (*n*)** : follow the best path; never expand anything else; difficult to compute **h(*n*)** in this way!
- **h(*n*) > d(*n*)** : not guarantee to compute a best path; but very fast
- **h(*n*) >> g(*n*)** : **h(*n*)** dominates -> A* becomes the Best First Search

A* Algorithm

- Add **START** to **OPEN** list
- while **OPEN** not empty
- get node *n* from **OPEN** that has the lowest f(*n*)
- if *n* is **GOAL** then return path
- move *n* to **CLOSED**
- for each *n*' = CanMove(*n*, direction)
-   g(*n*') = g(*n*) + 1
-   calculate h(*n*')
-   if *n*' in **OPEN** list and new *n*' is not better, continue
-   if *n*' in **CLOSED** list and new *n*' is not better, continue
-   remove any *n*' from **OPEN** and **CLOSED**
-   add *n* as *n*'s parent
-   add *n*' to **OPEN**
- end for
- end while
- if we get to here, then there is No Solution

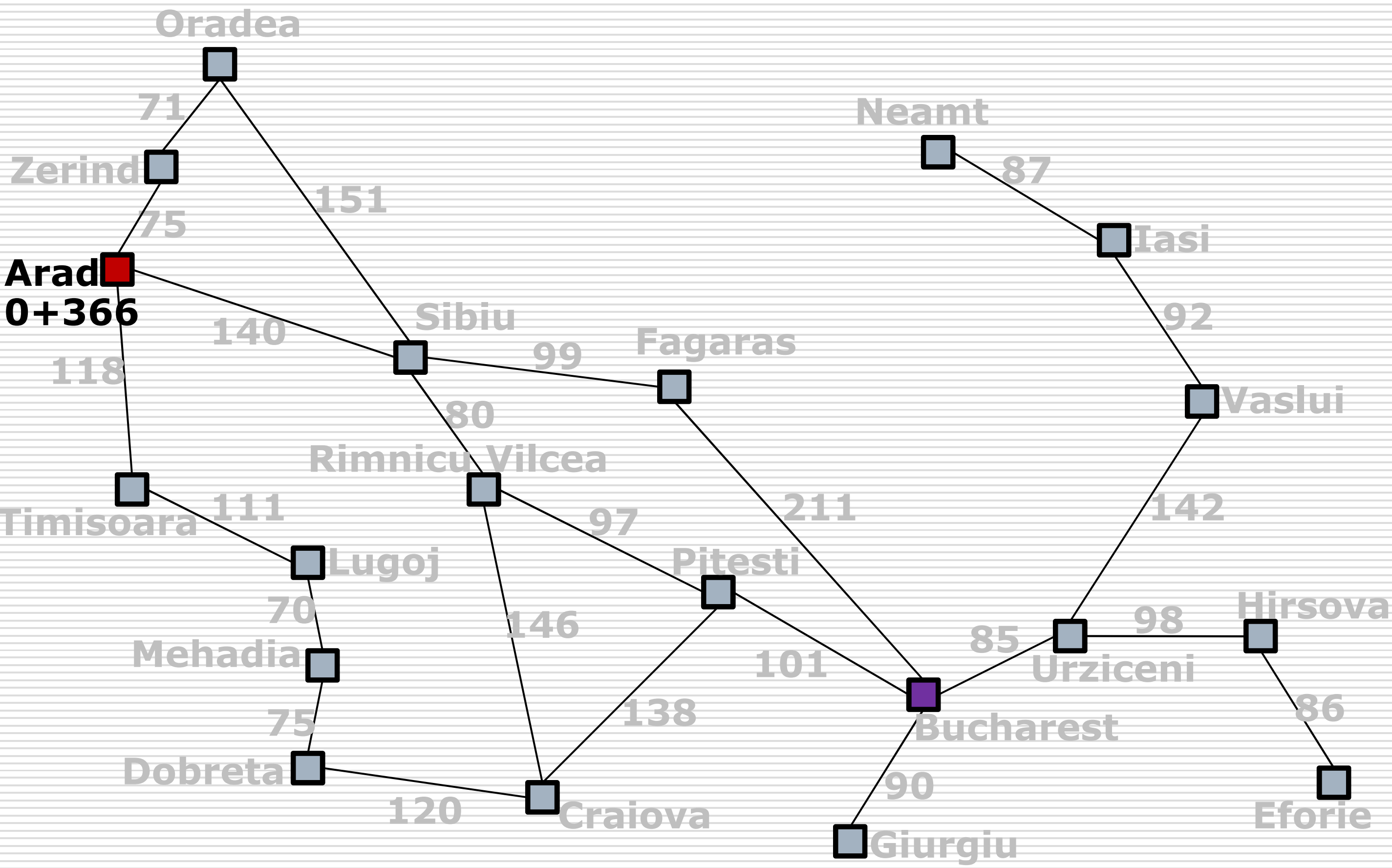
Example



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

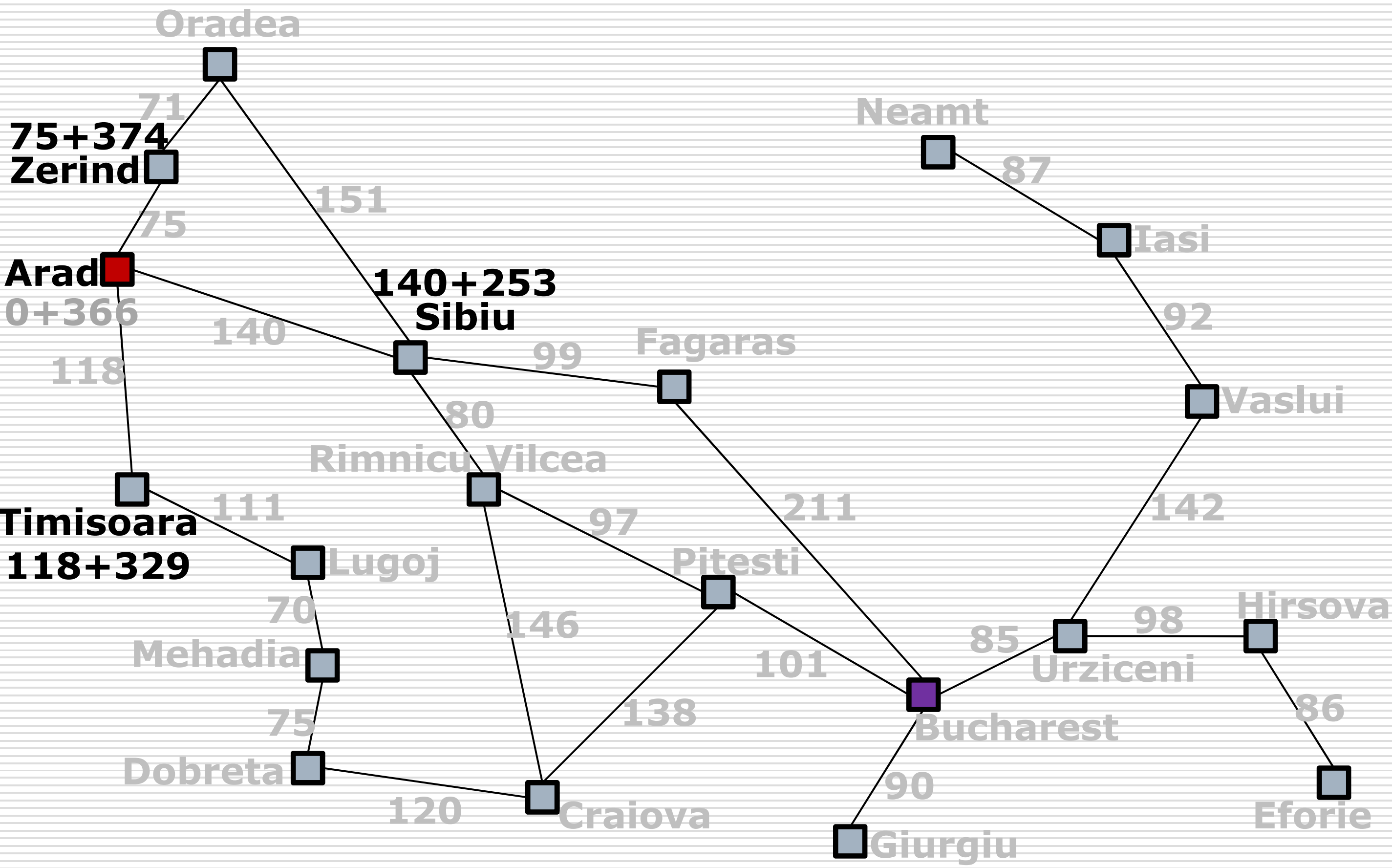
A* Example



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

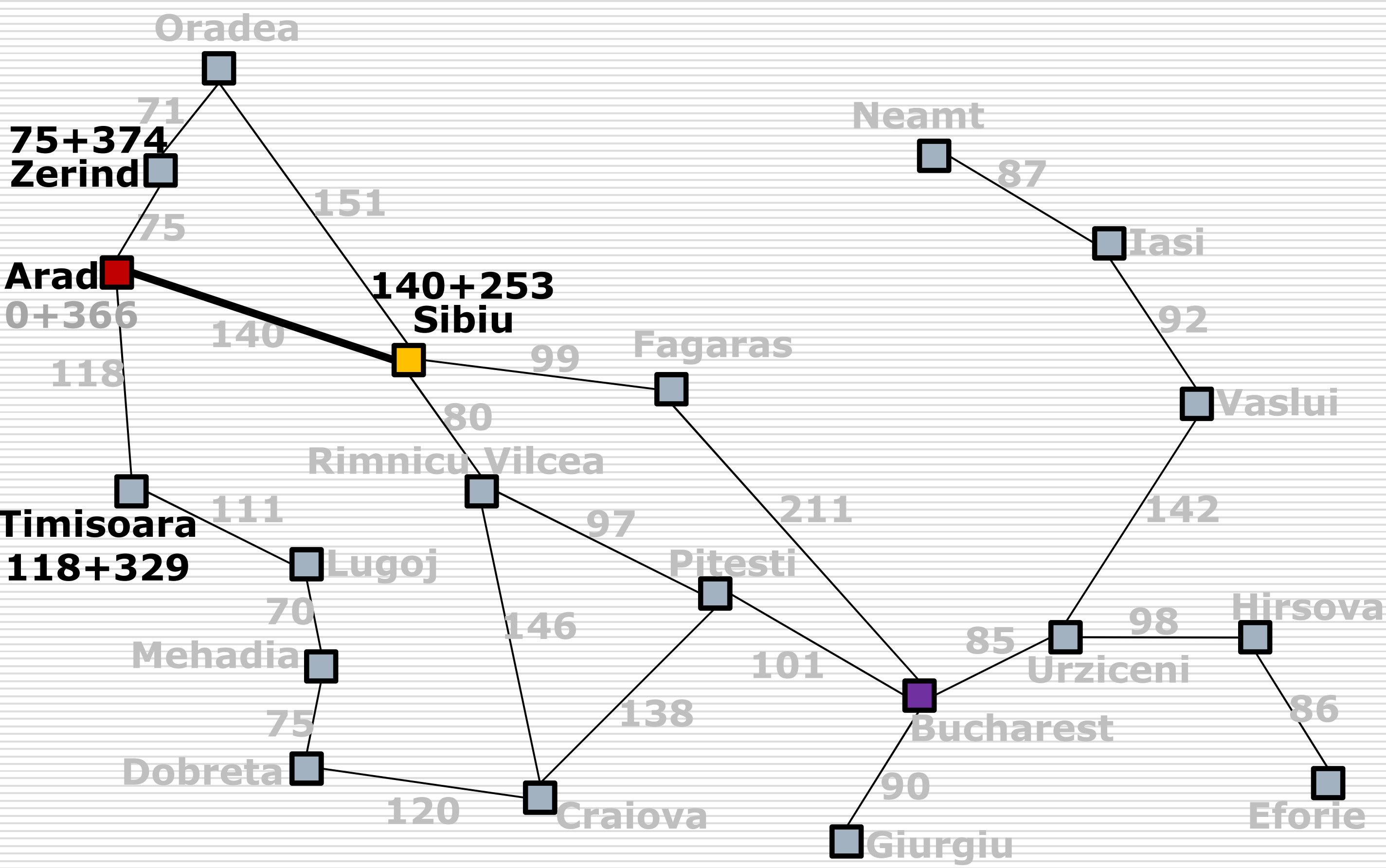
A* Example



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

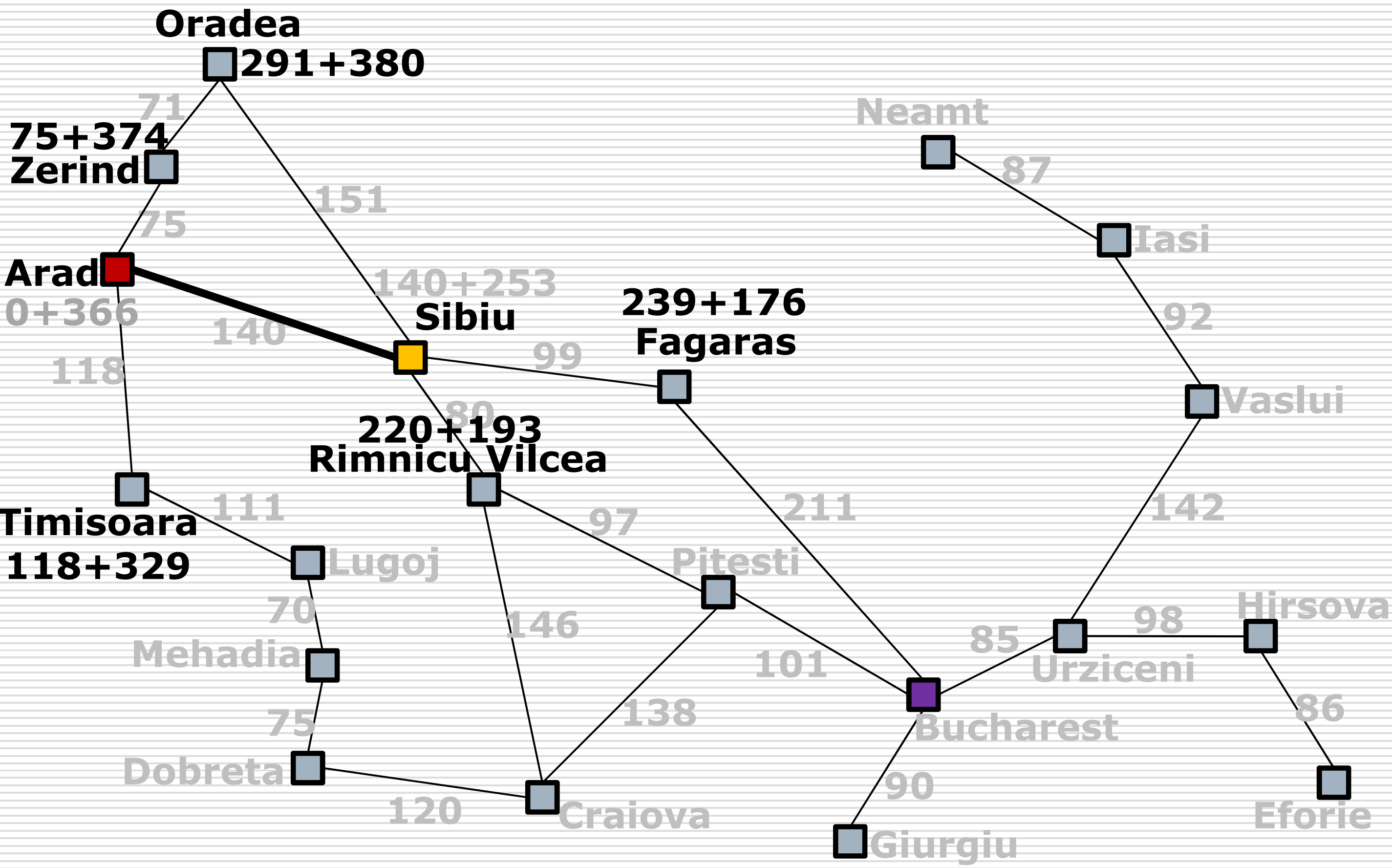
A* Example



Straight-line distance to Bucharest

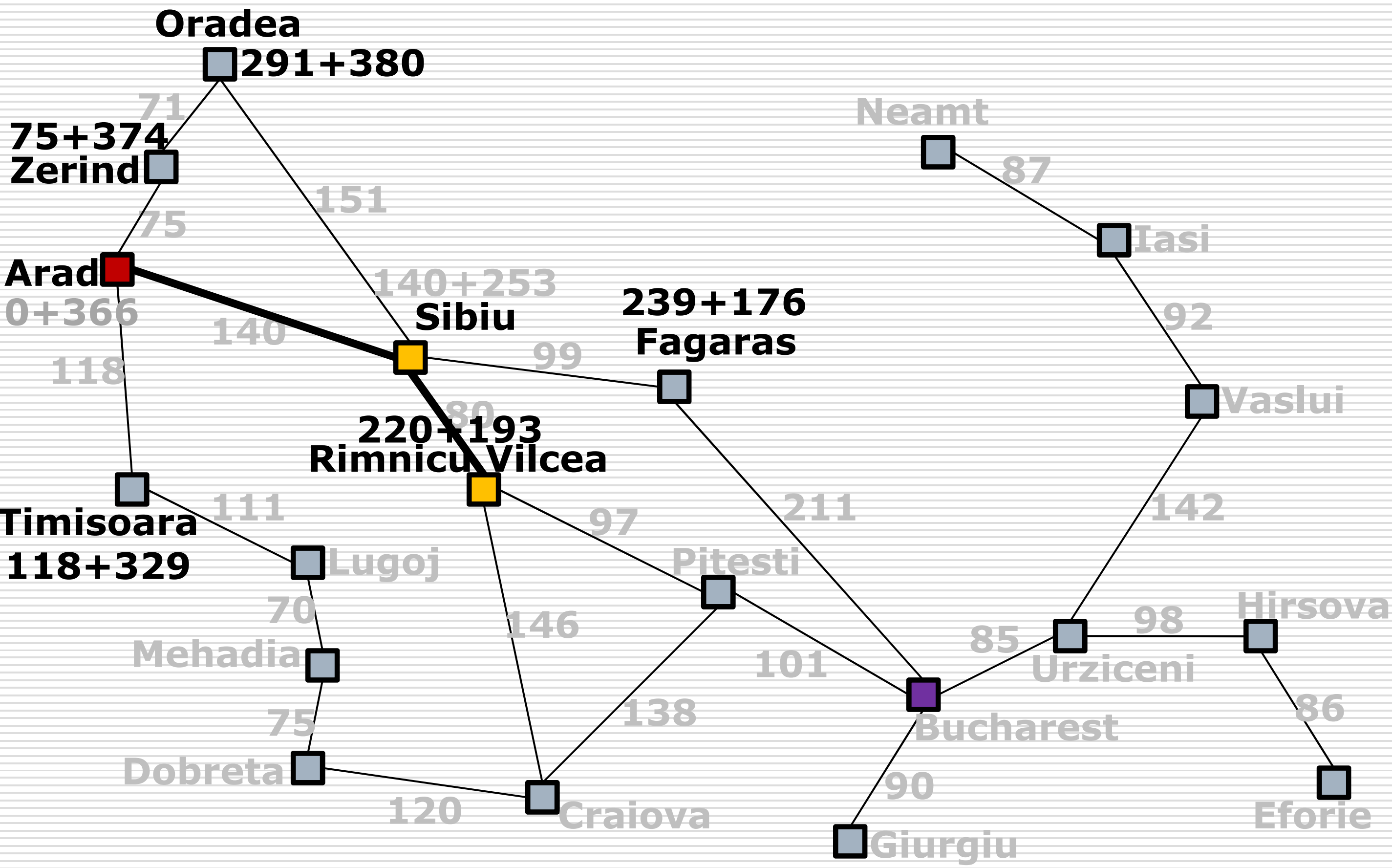
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



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

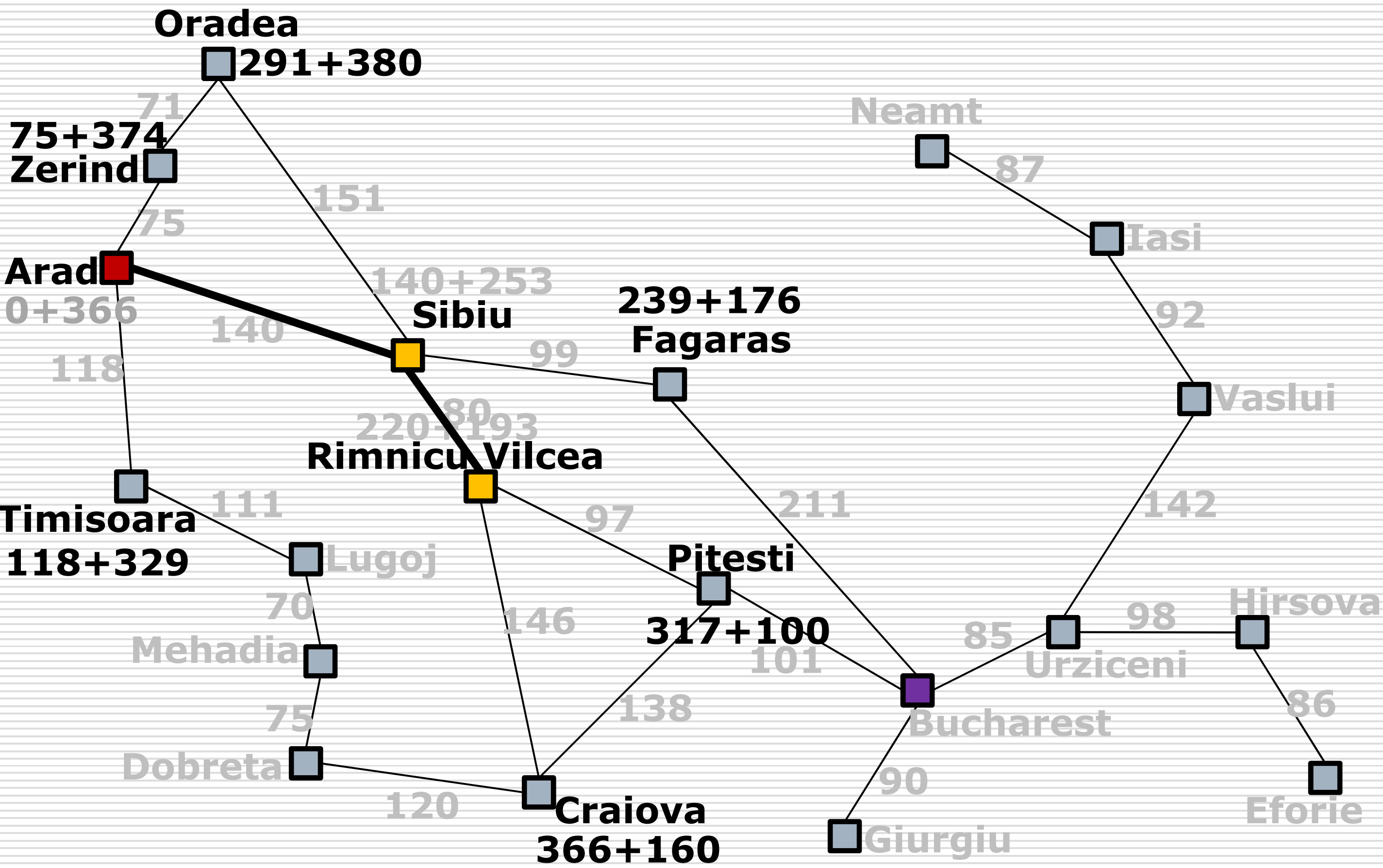
A* Example



Straight-line distance to Bucharest

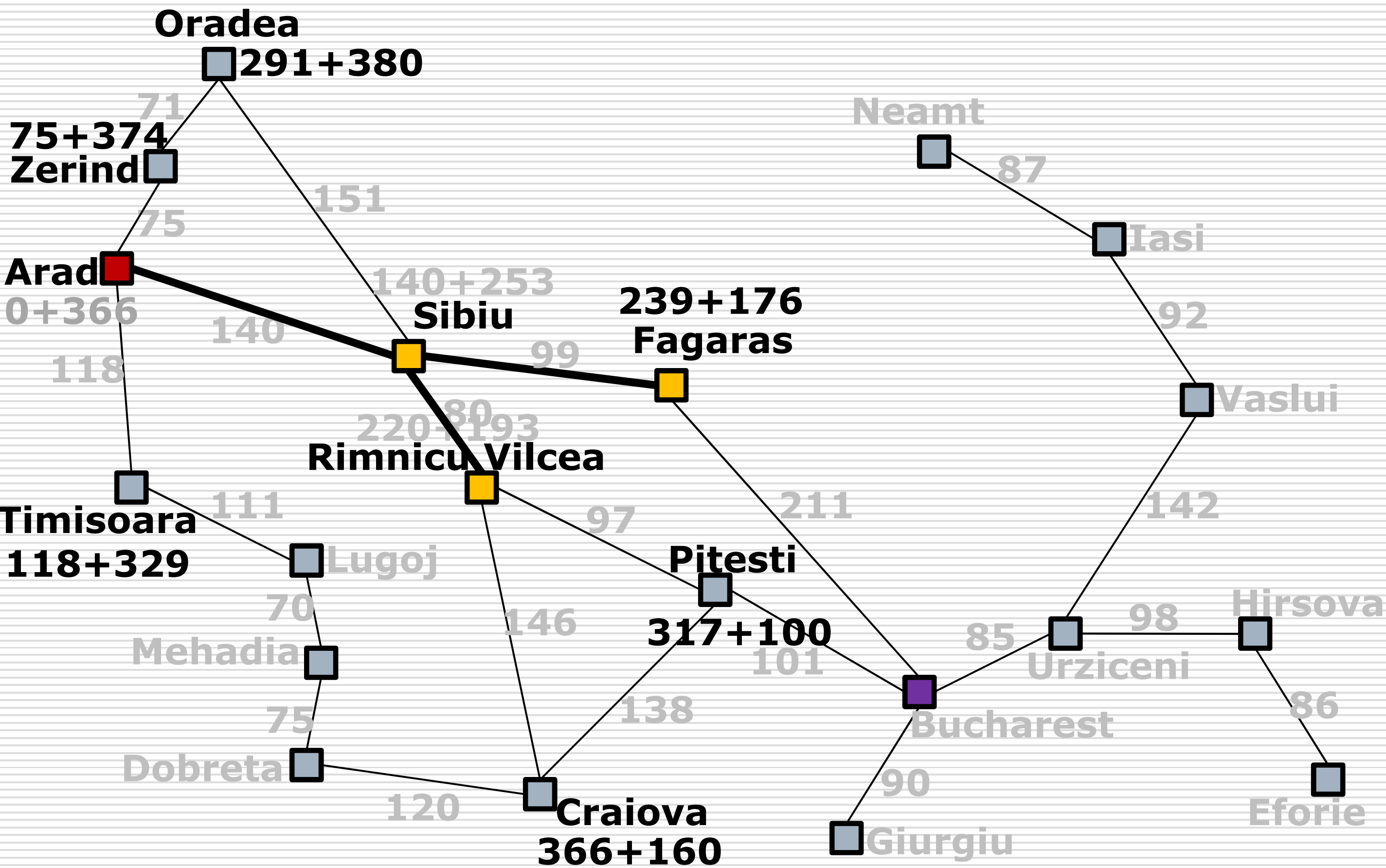
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



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

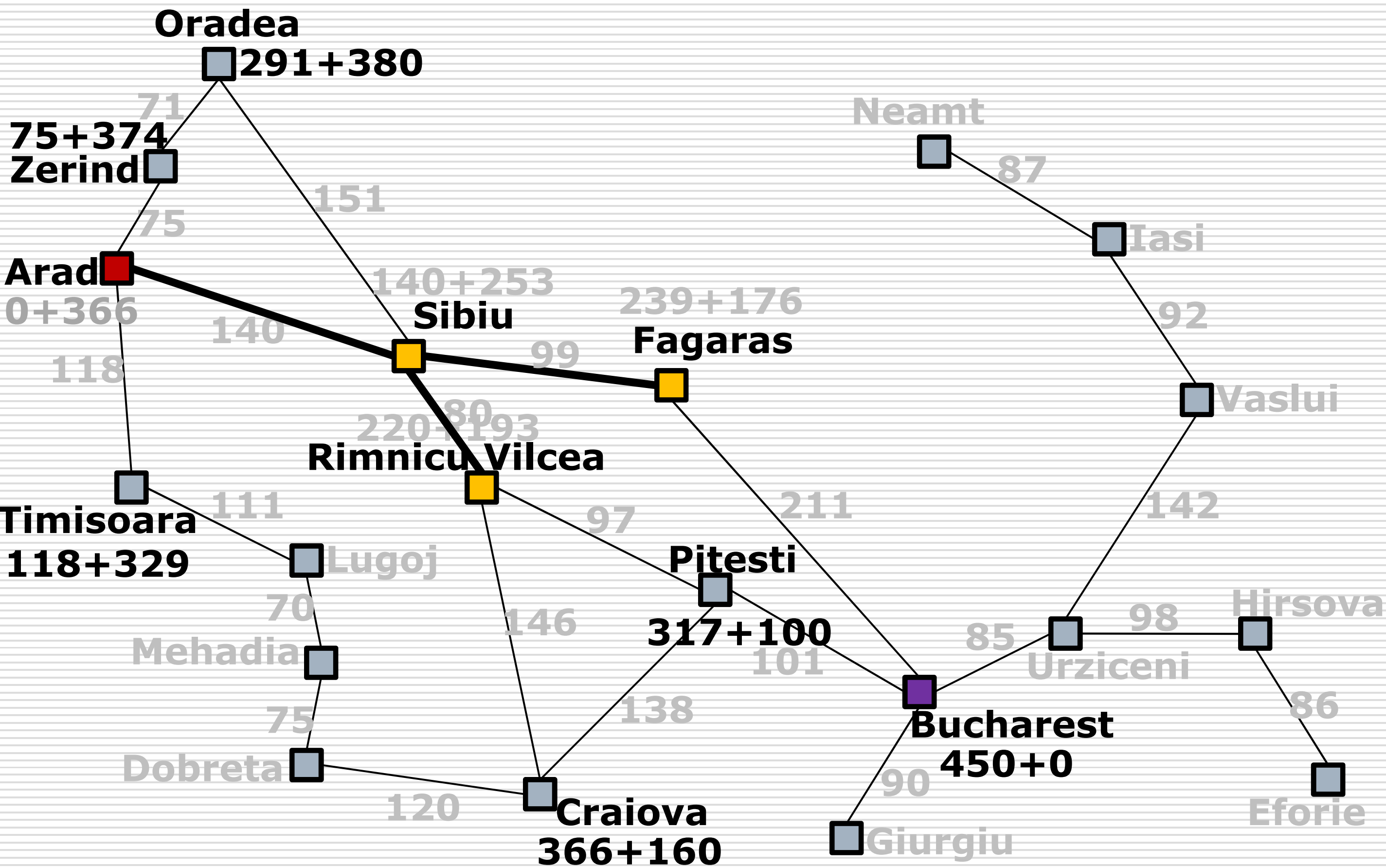
A* Example



Straight-line distance to Bucharest

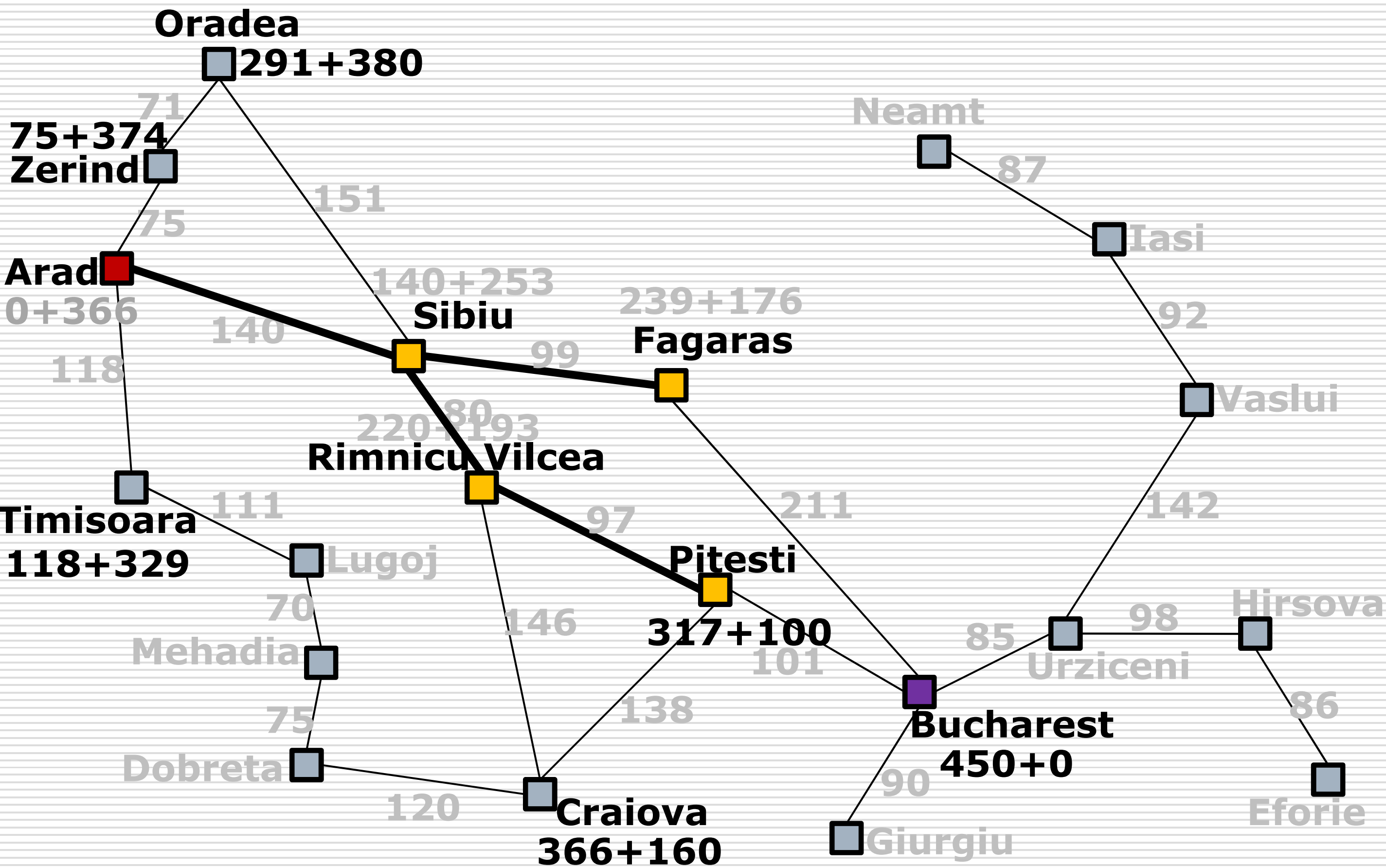
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



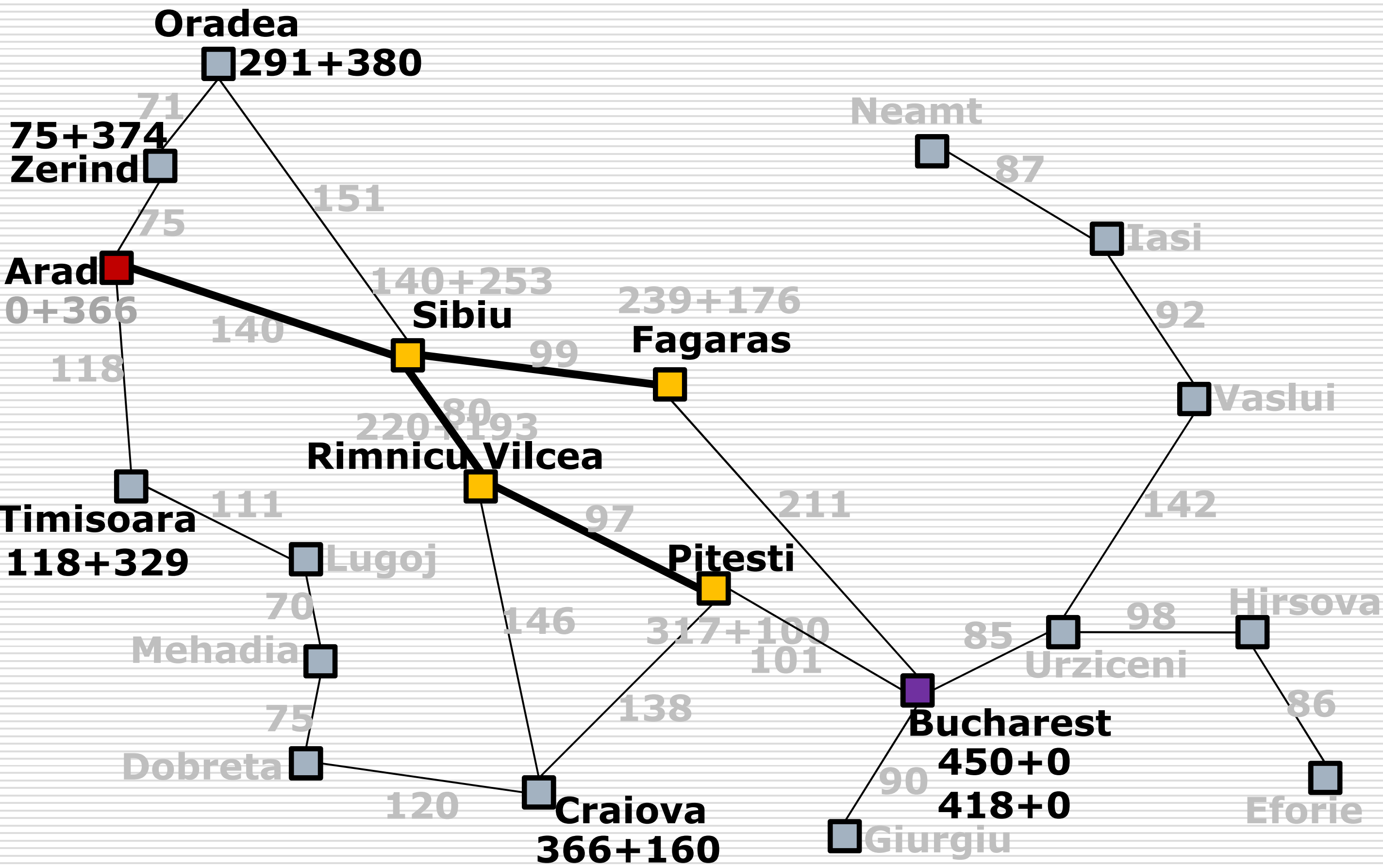
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



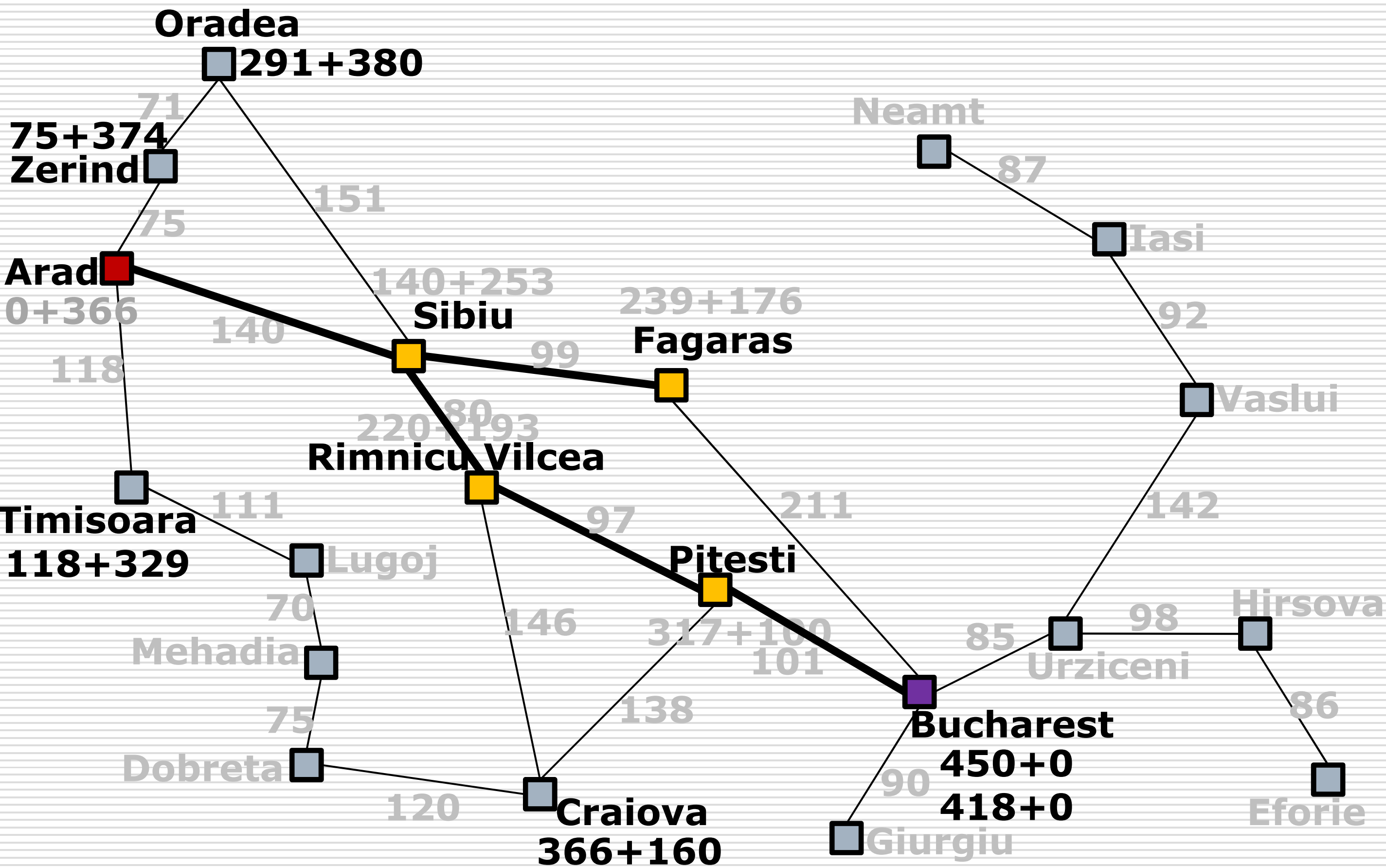
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



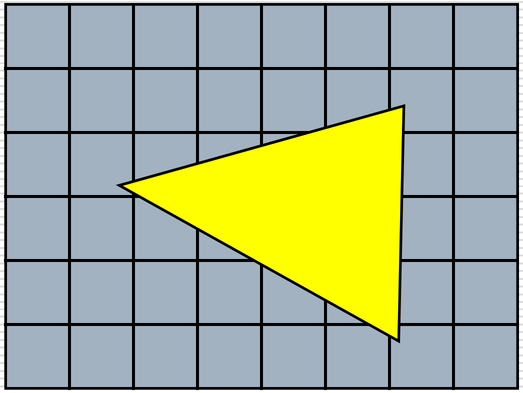
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Example



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

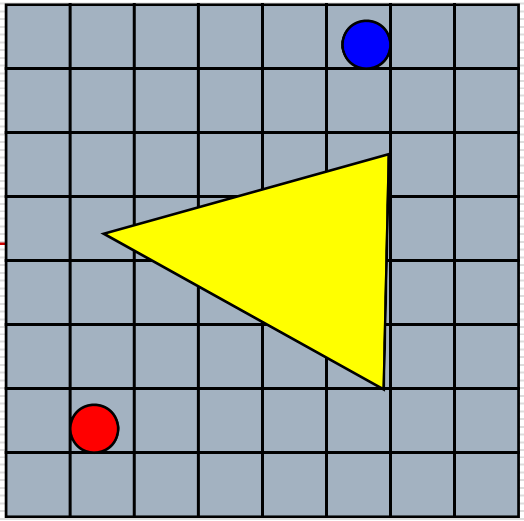
A* Algorithm



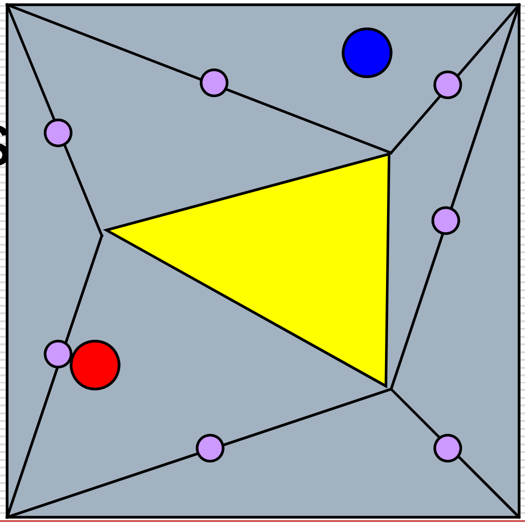
- State
  - Location
  - Neighboring states
- Search space
  - Related to terrain format
  - Grids
  - Triangles or Convex Polygons
  - Points of Visibility
- Cost estimate
- Path
  - Typical A* path
  - Straight path
  - Smooth path
- Hierarchical path finding

Search Space & Neighboring States

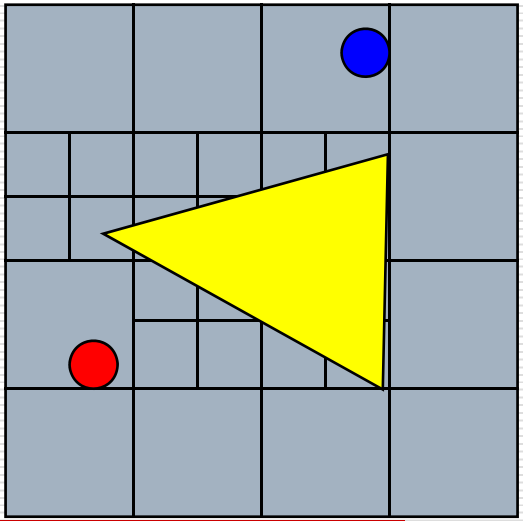
- Rectangular Grid
  - Use grid center
- Quadtree
  - Use grid center
- Triangles or Convex Polygons
  - Use edge mid-point
  - Use triangle center



Rectangular Grid



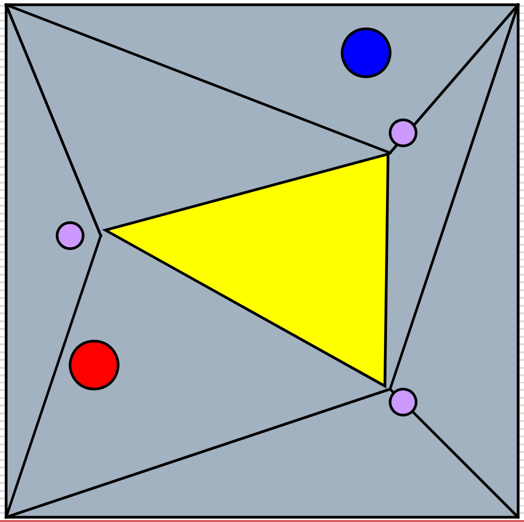
Triangles



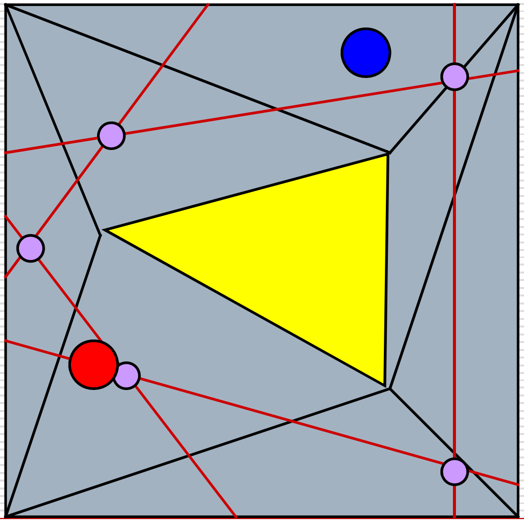
Quadtree

Search Space & Neighboring States

- Points of Visibility (POV)
- Generalized cylinders
  - Use intersections



Points of Visibility

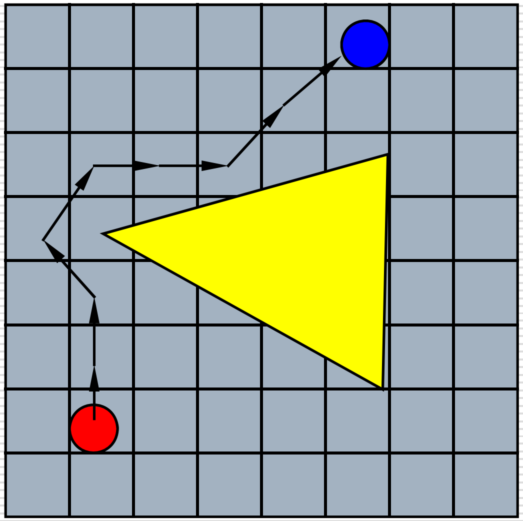


Generalized Cylinders

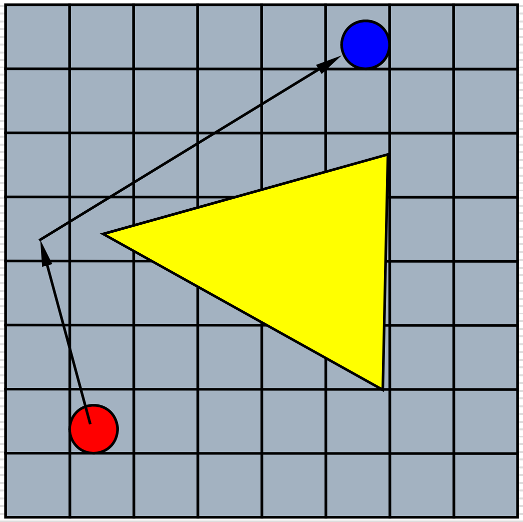
Cost Estimate

- Cost function
  - CostFromStart
  - CostToGoal
- Minimum cost
  - Distance traveled
  - Time of traveled
  - Movement points expended
  - Fuel consumed
  - Penalties for passing through undesired area
  - Bonuses for passing through desired area
  - ...
- Estimate
  - To goal “distance”

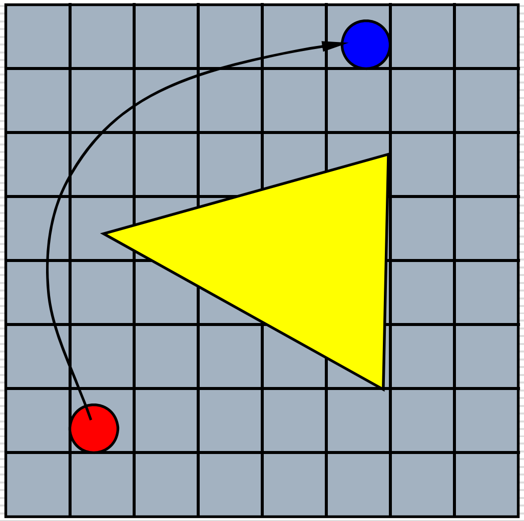
Result Path



Typical A* Path



Straight Path



Smooth Path
(Catmull-Rom Spline)

Hierarchical Path Finding

- Break the terrain for path finding to several ones hierarchically
  - Room-to-Room
  - 3D layered terrain
  - Terrain LOD
- Pros
  - Speedup the search
  - Solve the problem of layered path finding

Path Finding Challenges

- Moving Goal
  - Do you need to find path each frame ?
- Moving Obstacles
  - Prediction Scheme
- Complexity of the Terrain
  - Hierarchical path finding
- “Good” Path

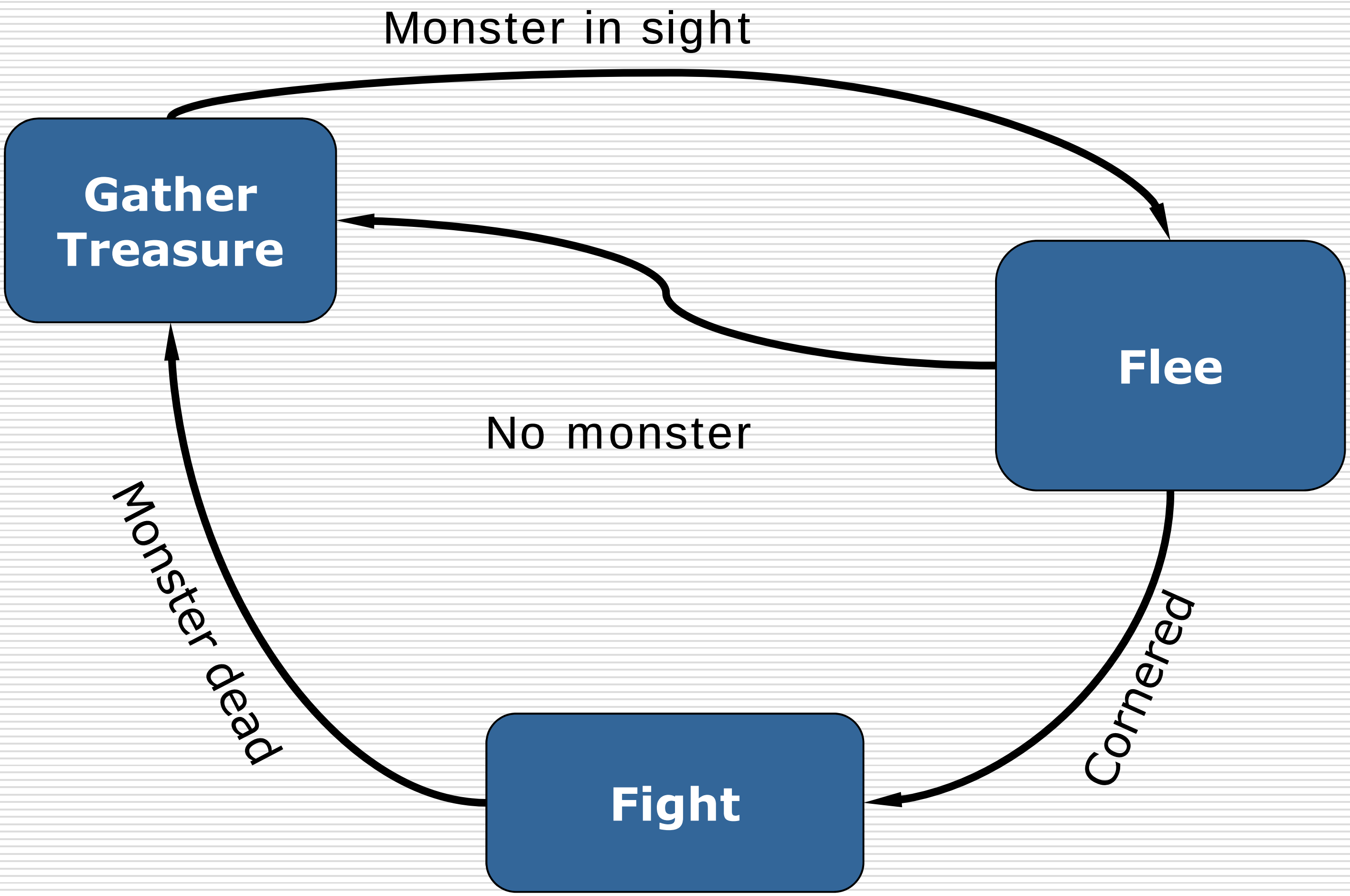
Introduction to FSM

- Finite State Machine (FSM) is the most commonly used game AI technology today.
  - Simple
  - Efficient
  - Easily extensible
  - Powerful enough to handle a wide variety of situations
- Theory (simplified)
  - A set of states, S
  - An input vocabulary, I
  - Transition function, T(s, i)
    - Map a state and an input to another state

Introduction to FSM

- Practical use
  - State
    - Behavior
  - Transition
    - Across states
    - Conditions
  - It's all about driving behavior
- Flow-chart diagram
  - UML State chart
    - Arrow
      - Transition
    - Rectangle
      - State

FSM Example


Monster in sight
Gather Treasure
Flee
No monster
Monster dead
Cornered
Fight

FSM for Games

- Character AI
- “Decision-Action” model
- Behavior
  - Mental state
- Transition
  - Players’ action
  - The other characters’ actions
  - Some features in the game world

Implement FSM

- Code-based FSM
  - Simple Code One Up
    - Straightforward
    - Most common
  - Macro-assisted FSM Language
- Data-Driven FSM
  - FSM Script Language

Coding an FSM – Code Example

```
void RunLogic(int *state) {
  switch(*state) {
    case 0:  // Gather Treasure
      GatherTreasure();
      if (SeeMonster()) *state = 1;
      break;
    case 1:  // Flee
      Flee();
      if (!SeeMonster()) *state = 0;
      if (Cornered()) *state = 2;
      break;
    case 2:  // Fight
      Fight();
      if (!SeeMonster()) *state = 0;
      break;
  }
}
```

FSM Language Use Macros

- Coding a state machine directly causes lack of structure
  - Going complex when FSM at their largest
- Use macros
- Beneficial properties
  - Structure
  - Readability
  - Debugging
- Simplicity

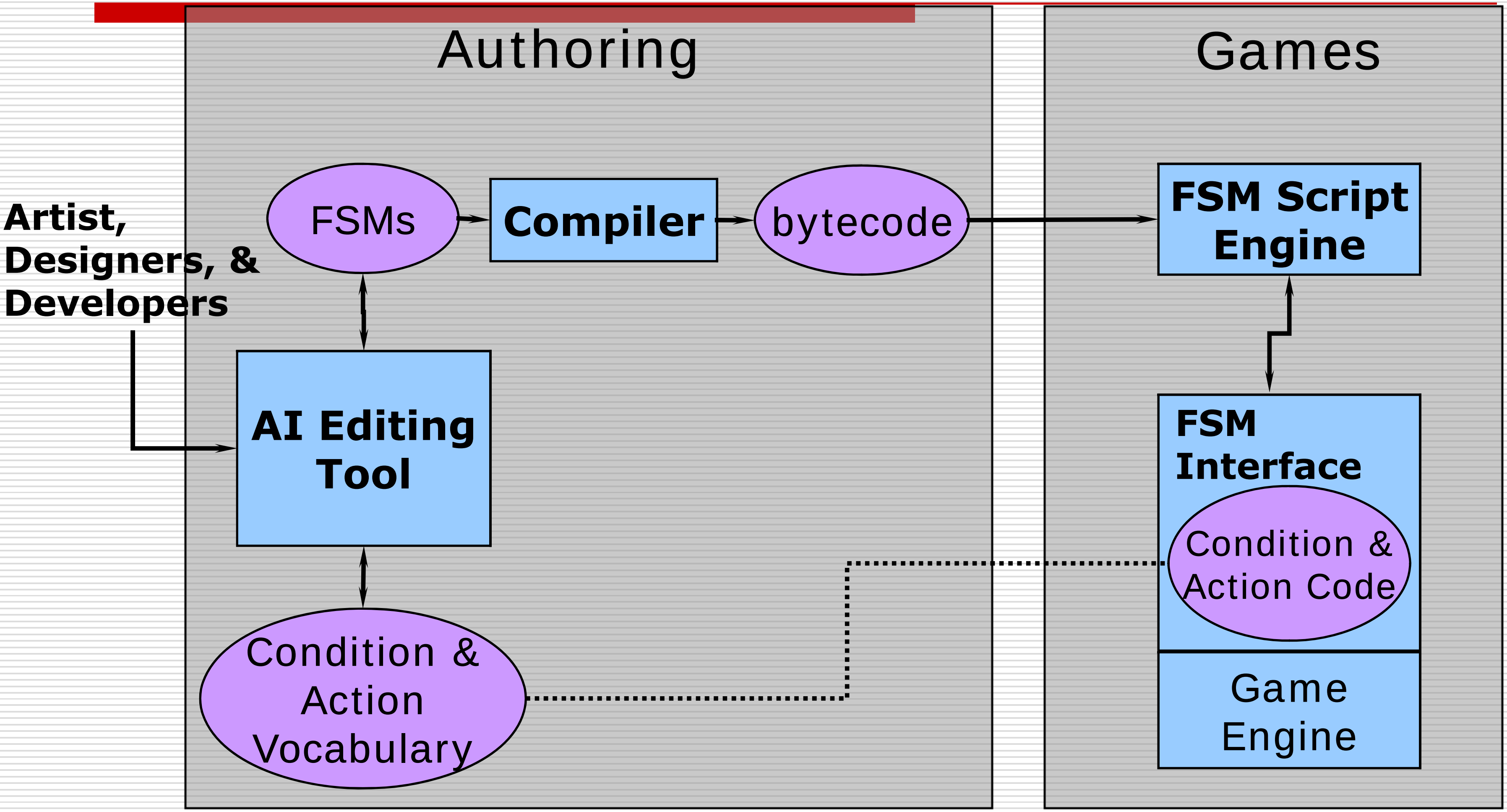
FSM Language Use Macros – An Example

```
#define BeginStateMachine ...
#define State(a) ...
...
bool MyStateMachine::States(StateMachineEvent event, int state) {
   BeginStateMachine
      State(0)
         OnUpdate
            GatherTreasure();
            if (SeeMonster()) SetState(1);
      State(1)
         OnUpdate
            Flee();
            SetState(0);
            if (!SeeMonster()) SetState(0);
            if (Cornered()) SetState(2);
      State(2);
         OnUpdate
            if (!SeeMonster()) SetState(0);
   EndStateMachine
}
```

Data-Driven FSM

- Scripting language
  - Text-based script file
  - Transformed into
    - C++
      - Integrated into source code
    - Bytecode
      - Interpreted by the game
- Authoring
  - Compiler
  - AI editing tool
- Game
  - FSM script engine
  - FSM interface

Data-Driven FSM Diagram



AI Editing Tool for FSM

- Pure text
  - Syntax ?
- Visual graph with text
- Used by Designers, Artists, or Developers
  - Non-programmers
- Conditions & action vocabulary
  - SeeMonster
  - Cornered
  - Fight
  - ...

FSM Interface

- Facilitating the binding between vocabulary and game world
- Glue layer that implements the condition & action vocabulary in the game world
- Native conditions
  - SeeMonster(), Cornered()
- Action library
  - Fight(...)

FSM Script Language Benefits

- Accelerated productivity
- Contributions from artists & designers
- Ease of use
- Extensibility

Processing Models for FSMs

- Processing the FSMs
  - Evaluate the transition conditions for current state
  - Perform any associated actions
- When and how ?
  - Depend on the exact need of games
- Three common FSM processing models
  - Polling
  - Event-driven
  - Multithread

Polling Processing Model

- Processing each FSM at regular time intervals
  - Tied to game frame rate
  - Or some desired FSM update frequency
  - Limit one state transition in a cycle
  - Give a FSM a time-bound
- Pros
  - Straightforward
  - Easy to implement
  - Easy to debug
- Cons
  - Inefficiency
    - Some transition are not necessary to check every frame
- Careful design to your FSM

Event-driven Processing Model

- Designed to prevent from wasted FSM processing
- An FSM is only processed when it's relevant
- Implementation
  - A Publish-subscribe messaging system (Observer pattern)
  - Allows the engine to send events to individual FSMs
  - An FSM subscribes only to the events that have the potential to change the current state
  - When an event is generated, the FSMs subscribed to that events are all processed
- "As-needed" approach
  - Should be much more efficient than polling ?
- Tricky balance for fine-grained or coarse-grained events

Multithread Processing Model

- Both polling & event-driven are serially processed
- Multithread processing model
  - Each FSM is assigned to its own thread for processing
  - Game engine is running in another separate thread
  - All FSM processing is effectively concurrent and continuous
  - Communication between threads must be thread-safe
    - Using standard locking & synchronization mechanisms
- Pros
  - FSM as an autonomous agent who can constantly and independently examine and react to his environment
- Cons
  - Overhead when many simultaneous characters active
  - Multithreaded programming is difficult

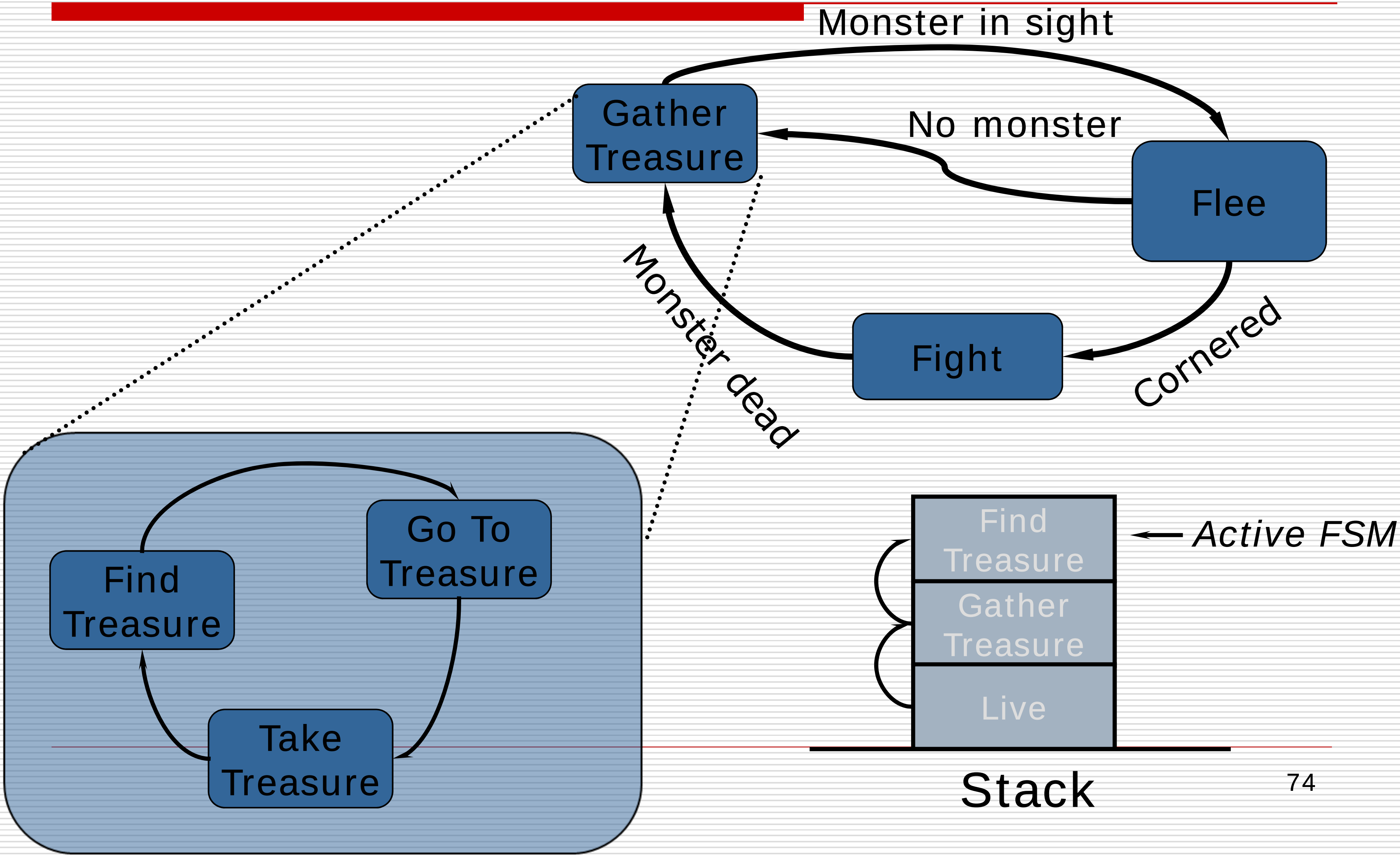
FSM Efficiency & Optimization

- Two categories :
  - Time spent
  - Computational cost
- Scheduled processing
  - Priority for each FSM
  - Different update frequency
- Load balancing scheme
  - Collecting statistics of past performance & extrapolating
- Time-bound for each FSM
- Do careful design
  - At the design level
- Level-of-detail FSMs

Level-Of-Detail FSMs

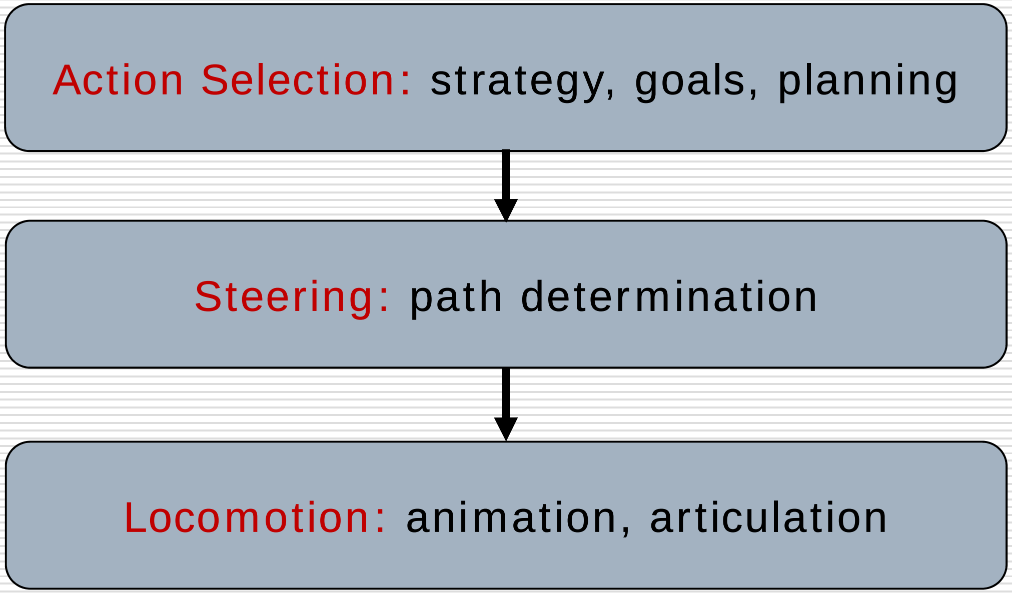
- Simplify the FSM when the player won't notice the differences
  - Outside the player's perceptual range
  - Just like the LOD technique used in 3D game engine
- Three design keys :
  - Decide how many LOD levels
    - How much development time available ?
    - The approximation extent
  - LOD selection policy
    - The distance between the NPC with the player ?
    - If the NPC can "see" the player ?
    - Be careful the problem of "visible discontinuous behavior"
  - What kind of approximations
    - Cheaper and less accurate solution

A Hierarchical FSM Example



Motion Behavior

- Action selection
- Steering
- Locomotion



A Hierarchy of Motion Behavior

Action Selection

- Game AI engine
  - State machine
    - Discussed in "Finite State Machine" section
  - Goals
  - Planning
  - Strategy
- Scripting
- Assigned by players
  - Players' input

Steering

- Path determination
  - Path finding or path planning
  - Discussed in “Path Finding”
- Behaviors
  - Seek & flee
  - Pursuit & evasion
  - Obstacle avoidance
  - Wander
  - Path following
  - Unaligned collision avoidance
- Group steering

Locomotion

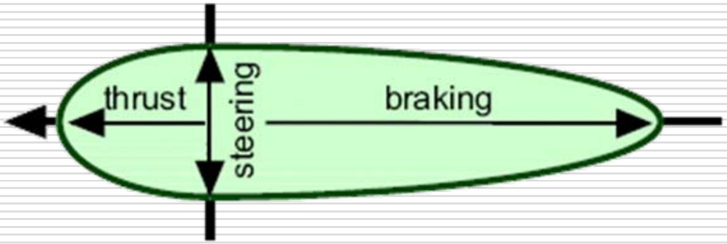
- Character physically-based models
- Movement
  - Turn right, move forward, ...
- Animation
  - By artists
- Implemented / managed by game engine

A Simple Vehicle Model

- A point mass
  - Linear momentum
  - No rotational momentum
- Parameters
  - Mass
  - Position
  - Velocity
    - Modified by applied forces
    - Max speed
      - Top speed of a vehicle
  - Max steering force
    - Self-applied
  - Orientation
    - Car
    - Aircraft

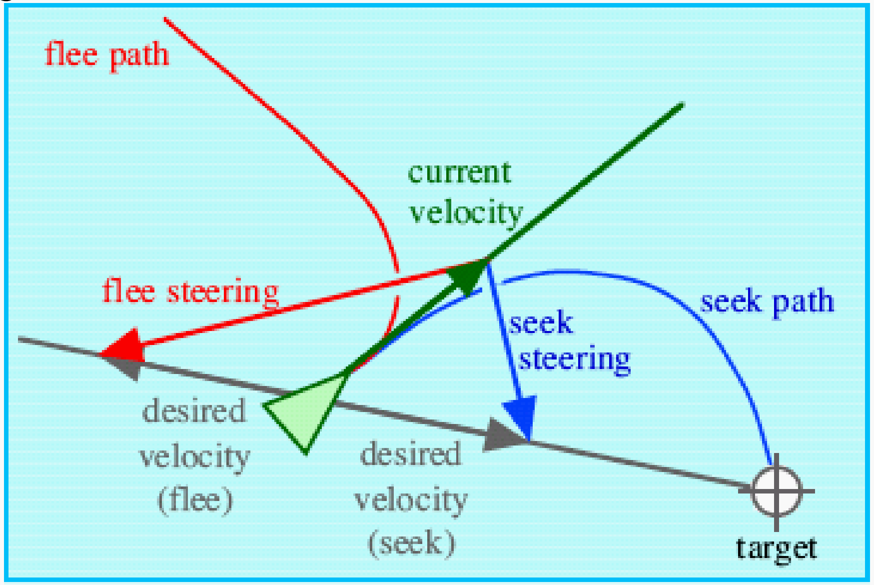
A Simple Vehicle Model

- Local space
  - Origin
  - Forward
  - Up
  - Side
- Steering forces
  - Asymmetrical
    - Thrust
    - Braking
    - Steering
- Velocity alignment
  - No slide, spin, ...
  - Turn



Seek & Flee Behaviors

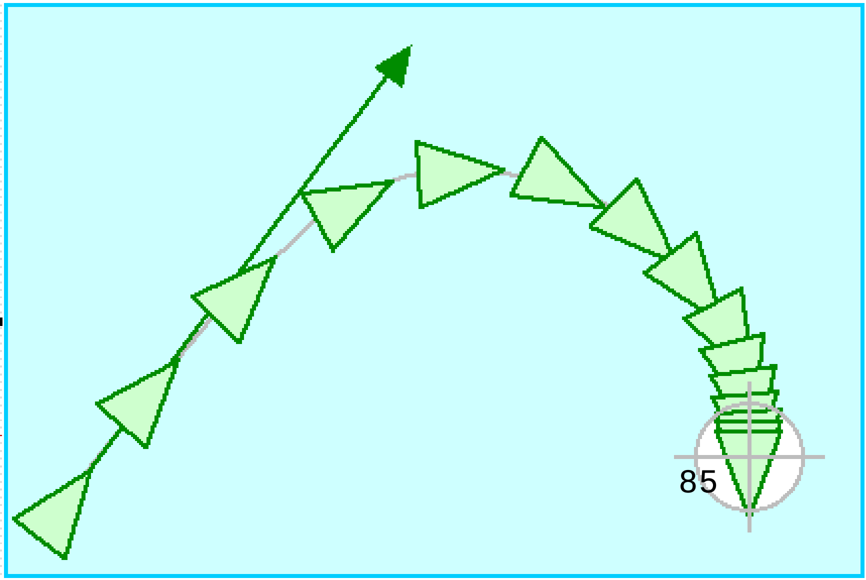
- Pursuit to a static target
  - Steer a character toward to a target position
- "A moth buzzing a light bulb"
- Flee
  - Inverse of seek
- Variants
  - Arrival
  - Pursuit to a moving target



- Seek Steering force
  - desired_velocity = normalize(target - position)*max_speed
  - steering = desired_velocity – velocity

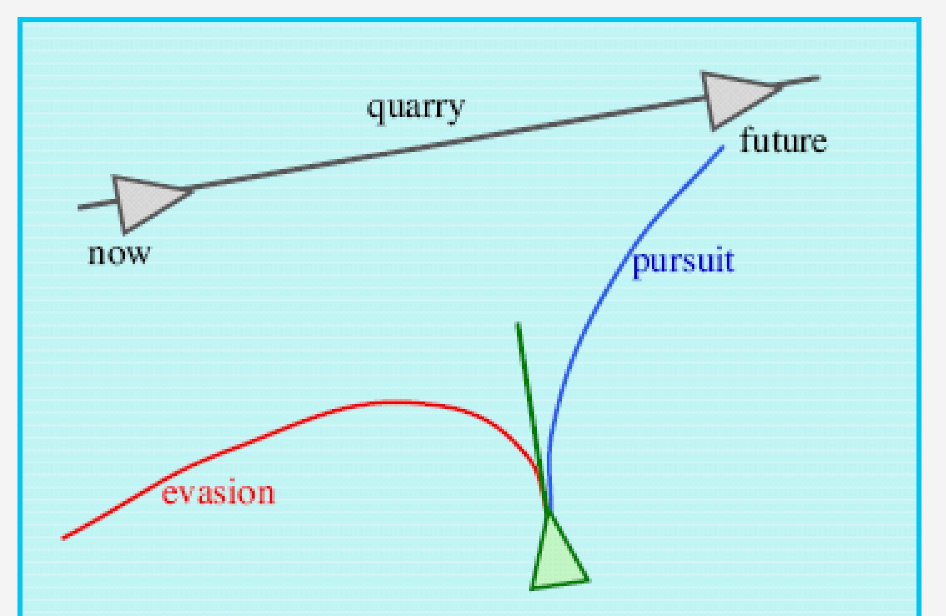
Arrival Behavior

- Identical to “Seek” while the character is far from its target
- Slow down as approaching the target, eventually slowing to a stop coincident with the target
- The desired velocity is clipped to max_speed outside the stopping radius, and inside it is ramped down (e.g. linearly) to zero.



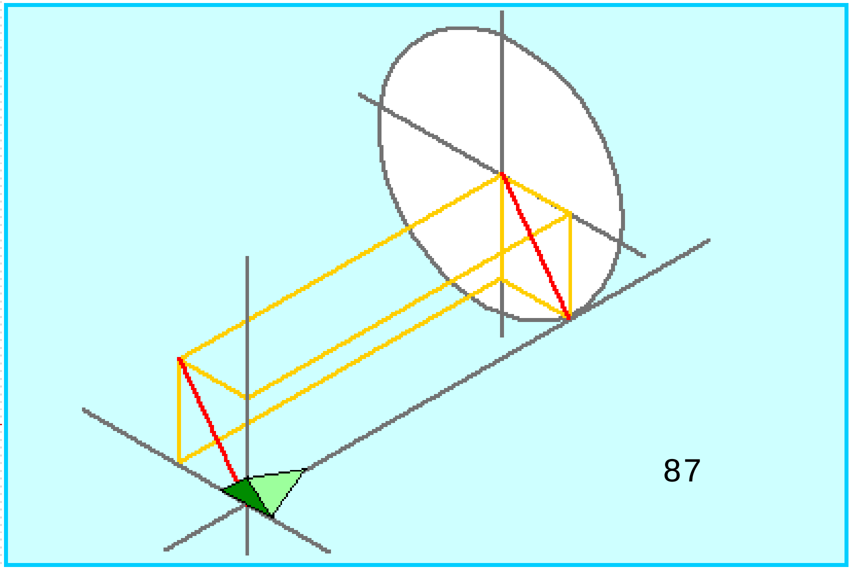
Pursuit & Evasion Behaviors

- Target is moving
- Apply seek or flee to the target's predicted position
- Estimate the prediction interval T
  - T = Dc
  - D = distance(pursuit, quarry)
  - c = turning parameter
- Variants
  - Offset pursuit
    - "Fly by"



Offset Pursuit Behavior

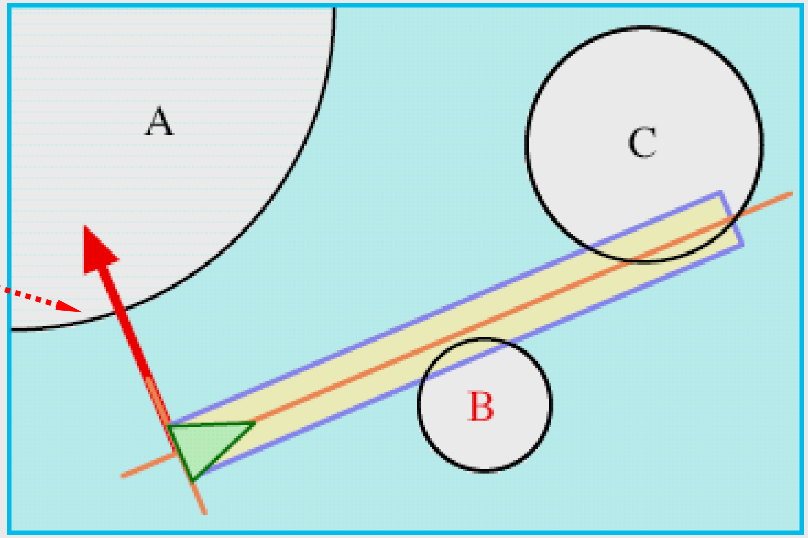
- Passes near, but not directly into a moving target
- Flying near enough to be within weapon range without colliding with the target
- Compute a target point given a radius R from the target's predicted position, and seek the point



Obstacle Avoidance Behavior

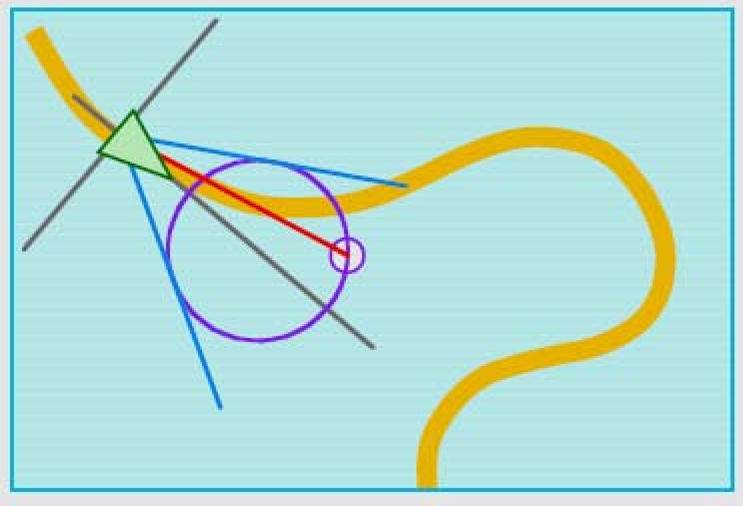
- Use bounding sphere
- Not collision detection
- Probe
  - A cylinder lying along forward axis
  - Diameter = character's bounding sphere
  - Length = speed (means Alert range)
- Find the most threaten obstacle
  - Nearest intersected obstacle
- Steering

steering force



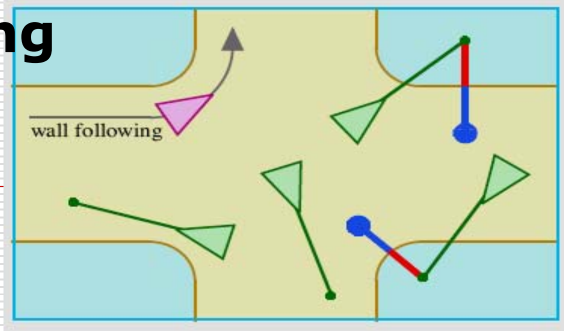
Wander Behavior

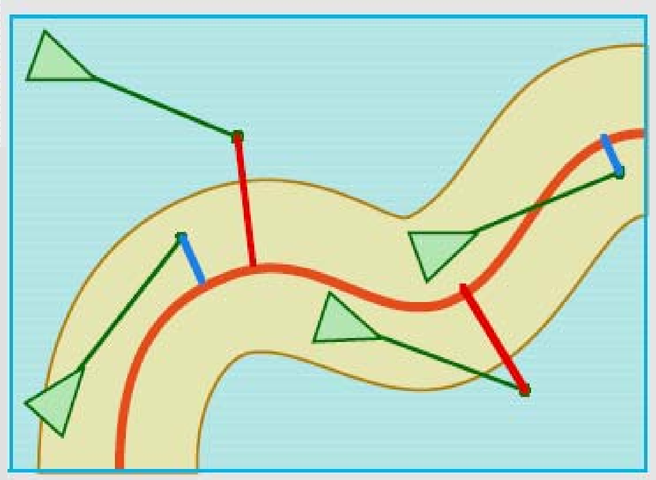
- Random steering
- One solution :
  - Retain steering direction state
    - Constrain steering force to the sphere surface located slightly ahead of the character
  - Make small random displacements to it each frame
    - A small sphere on sphere surface to indicate and constrain the displacement
- Another one :
  - Perlin noise
- Variants
  - Explore



Path Following Behavior

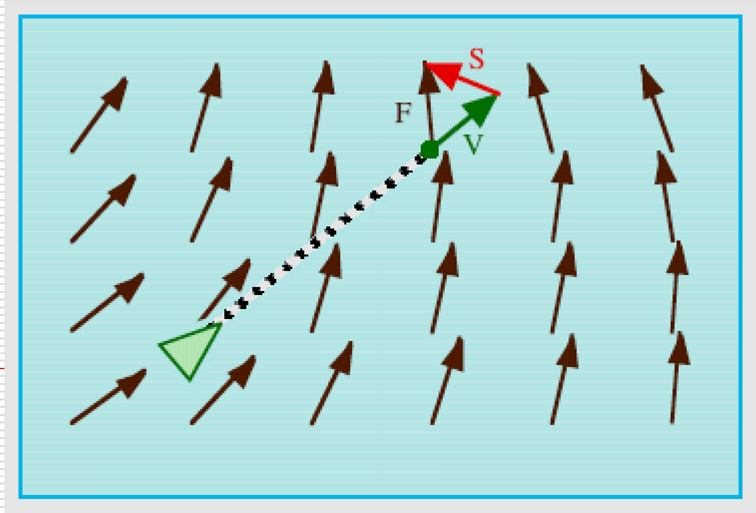
- The path
  - Spine
    - A spline or poly-line to define the path
  - Pipe
    - The tube or generated cylinder by a defined “radius”
- Following
  - A velocity-based prediction position
    - Inside the tube
      - Do nothing about steering
    - Outside the tube
      - “Seek” to the on-path projection
- Variants
  - **Wall following**
  - Containment





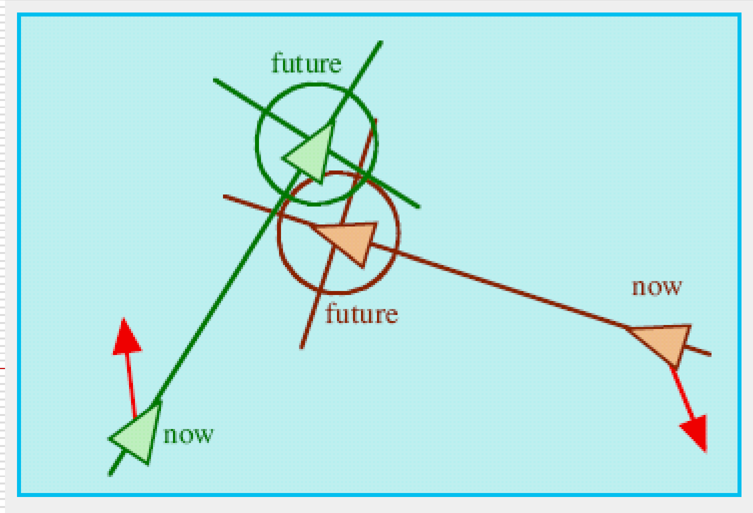
Flow Field Following Behavior

- A flow field environment is defined.
- Virtual reality
  - Not common in games



Unaligned Collision Avoidance Behavior

- Turn away from possible collision
- Predict the potential collision
  - Use bounding spheres
- If possibly collide,
  - Apply the steering on both characters
  - Steering direction is possible collision result
    - Use “future” possible position
    - The connected line between two sphere centers

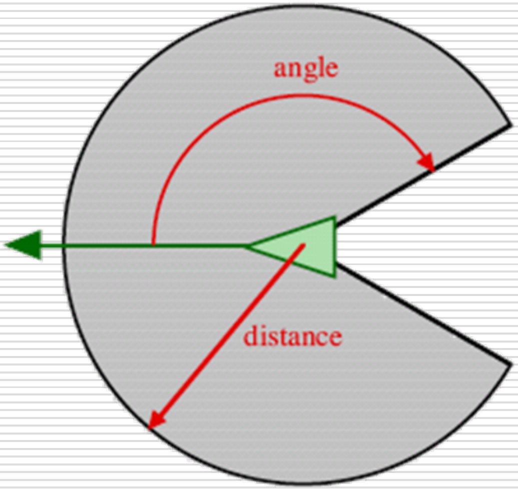


Steering Behaviors for Groups of Characters

- Steering behaviors determining how the character reacts to the other characters within his/her local neighborhood
- The behaviors including :
  - Separation
  - Cohesion
  - Alignment

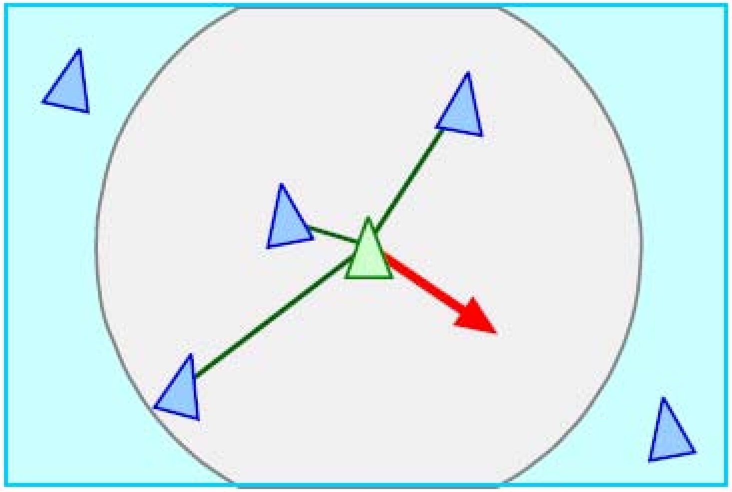
The Local Neighborhood of a Character

- The local neighborhood is defined as :
  - A distance
  - The field-of-view
    - Angle



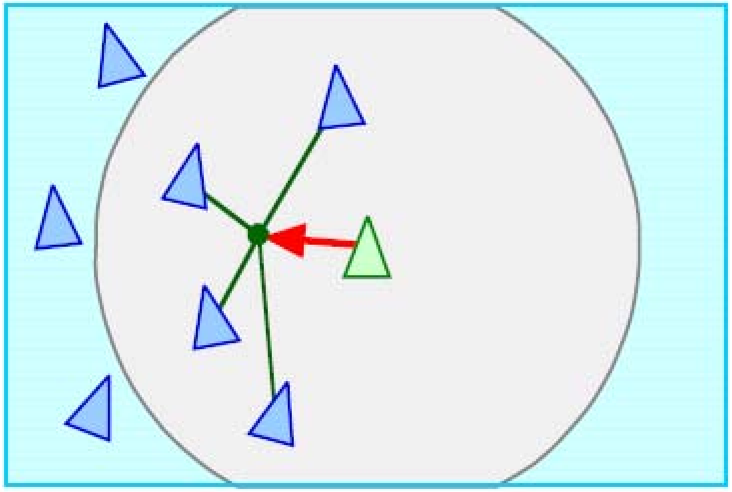
Separation Behavior

- Make a character to maintain a distance from others nearby.
  - Compute the repulsive forces within local neighborhood
    - Calculate the position vector for each nearby
    - Normalize it
    - Weight the magnitude with distance
      - 1/distance
    - Sum the result forces
    - Negate it



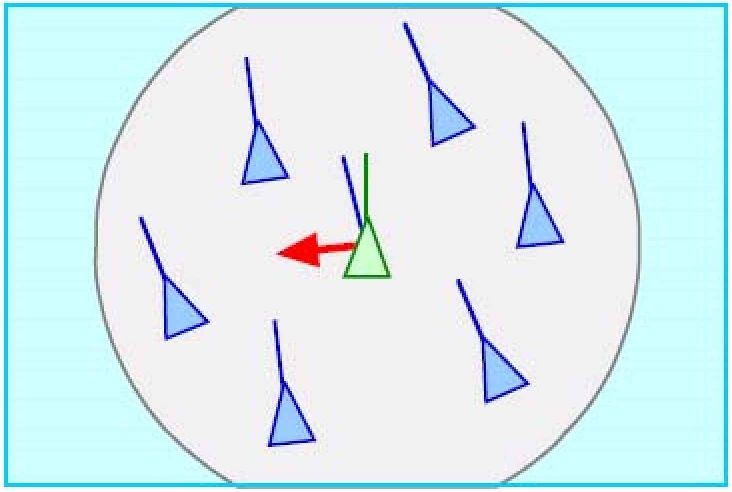
Cohesion Behavior

- Make a character to cohere with the others nearby
  - Compute the cohesive forces within local neighborhood
    - Compute the average position of the others nearby
      - Gravity center
    - Apply "Seek" to the position



Alignment Behavior

- Make a character to align with the others nearby
  - Compute the steering force
    - Average the together velocity of all other characters nearby
    - The result is the desired velocity
    - Correct the current velocity to the desired one with the steering force

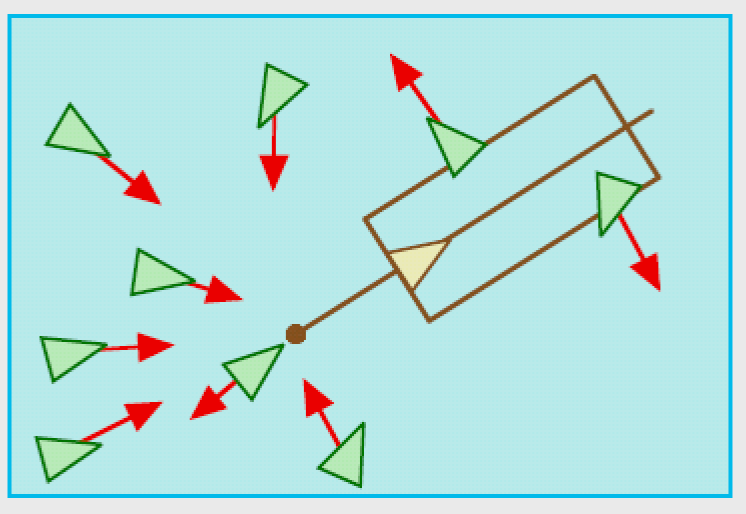


Flocking/Crowd Behavior

- “Boids Model of Flocks”
  - [Reynolds 87]
- Combination of :
  - Separation steering
  - Cohesion steering
  - Alignment steering
- For each combination including :
  - A weight for each combination
  - A distance
  - An Angle

Leader Following Behavior

- Follow a leader
  - Stay with the leader
    - "Pursuit" behavior (Arrival style)
  - Stay out of the leader's way
    - Defined as "next position" with an extension
    - "Evasion" behavior when inside the above area
  - "Separation" behavior for the followers



Behavior Conclusion

- A simple vehicle model with local neighborhood
- Common steering behaviors including :
  - Seek
  - Flee
  - Pursuit
  - Evasion
  - Offset pursuit
  - Arrival
  - Obstacle avoidance
  - Wander
  - Path following
  - Wall following
  - Containment
  - Flow field following
  - Unaligned collision avoidance
  - Separation
  - Cohesion
  - Alignment
  - Flocking
  - Leader following

More Topics in Game AI

- Scripting
- Goal-based planning
- Rule-based inference engine
- Neural network
- References
  - Game Programming Gems
  - AI Game Programming Wisdom