Introduction to Game AI

Behrouz Minaei

Iran University of Science and Technology

Inspirations

Marvin Minsky
Different representations for different views

Noam Chomsky
Hierarchical decomposition
Structure vs Meaning
“Colorless green ideas sleep furiously”

Damian Isla
Cognitive Maps
Spatial Relations
Semantics

Daniel Dennet
Behaviour can be viewed at the physical, design and intentional levels

Chris Hecker
Style vs Structure [2008]
What is the textured triangle of A.I.

Craig Reynolds
Simple rules
Complex behaviour

Structure of an AI system

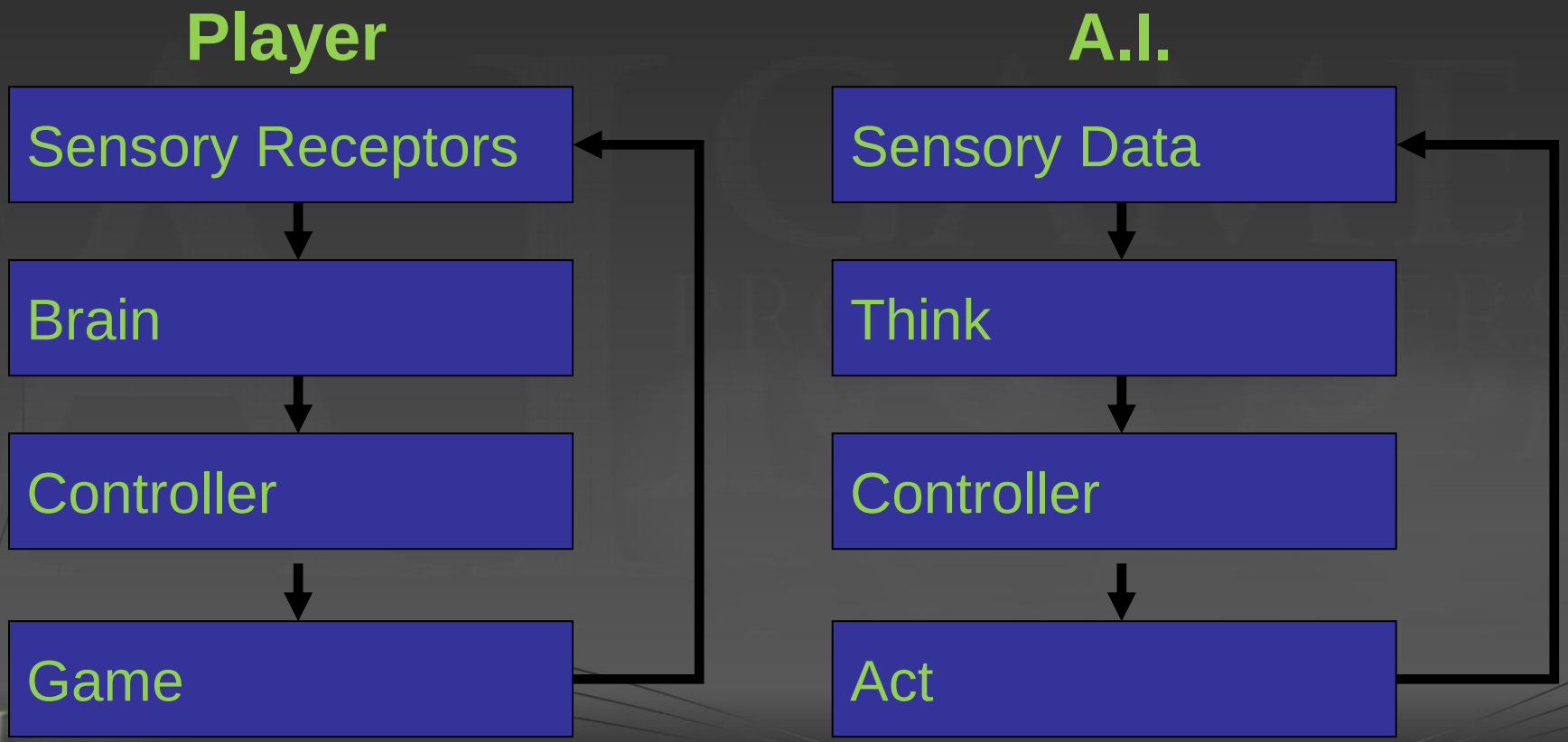
- Fundamentally, AI systems come in two flavors.
  - The first and most common is the agent, which is a virtual character in the game world.
  - These are usually enemies, but can also be nonplaying characters, sidekicks, or even an animated cow in a field.
  - Thus, these AI systems are structured in a way similar to our brain. It is easy to identify four elements or rules:
    - A sensor or input system
    - A working memory
    - A reasoning/analysis core
    - An action/output system

AI

Structure of an AI system

- The second type of AI entity is abstract controllers.
- Take a strategy game, for example. Who provides the tactical reasoning?
- Each unit might very well be modeled using the preceding rules, but clearly, a strategy game needs an additional entity that acts like the master controller of the CPU side of the battle.
- This is not an embodied character but a collection of routines that provide the necessary group dynamics to the overall system. Abstract controllers have a structure quite similar to the one explained earlier, but each subsystem works on a higher level than an individual.

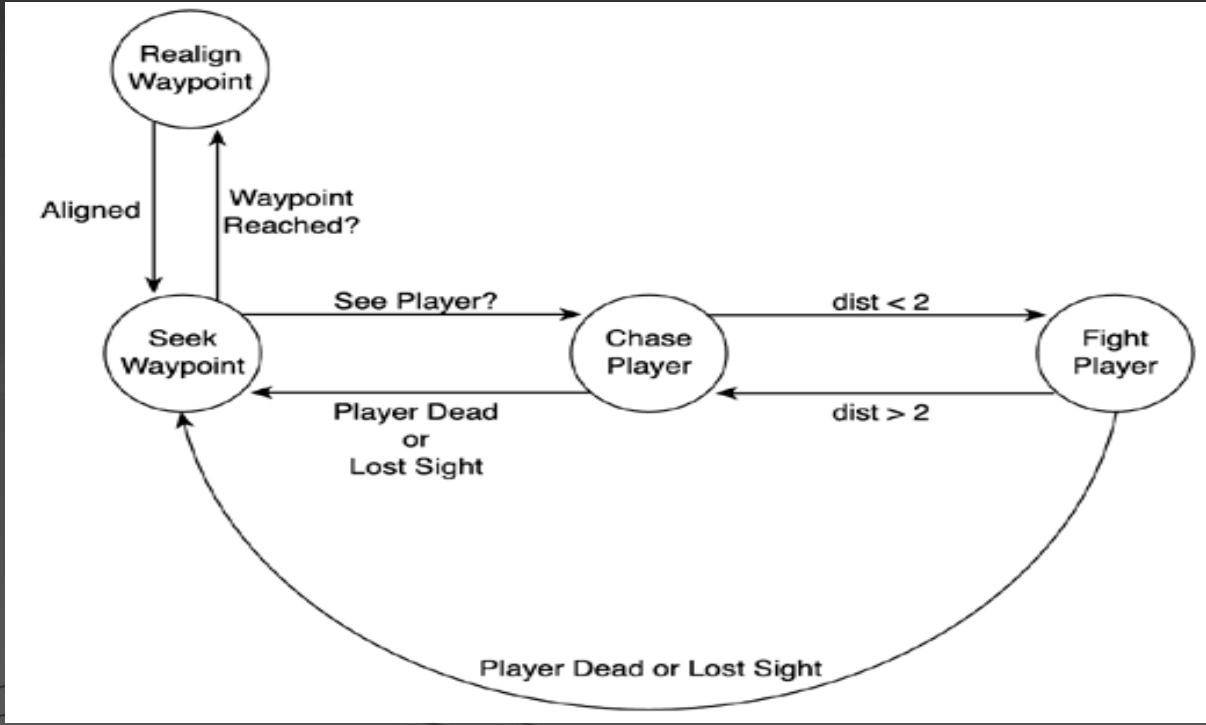
What we are looking for?



Game AI Techniques

- Finite State Machines
- It is a good idea to write down the primitive behaviors you need your AI to perform.
- We will then represent these graphically to lay them down into actual running code.
- Let's say we have these elements:
  - The enemy is in an outdoors area with no obstacles.
  - He has a predefined set of waypoints he patrols in a cyclical way.
  - The enemy activates when you get inside his viewing cone.
  - If he sees you, he chases you around.
  - He carries a sword.
  - If in close contact, he will remain stopped, hitting you with the sword.

Game AI Technique-FSM



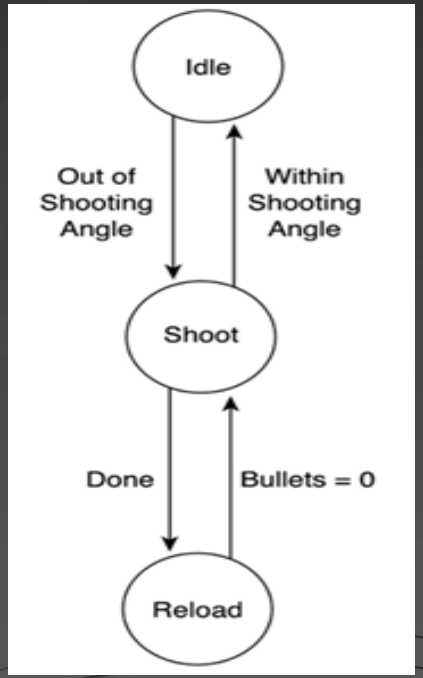
Game AI Techniques-Parallel Automata

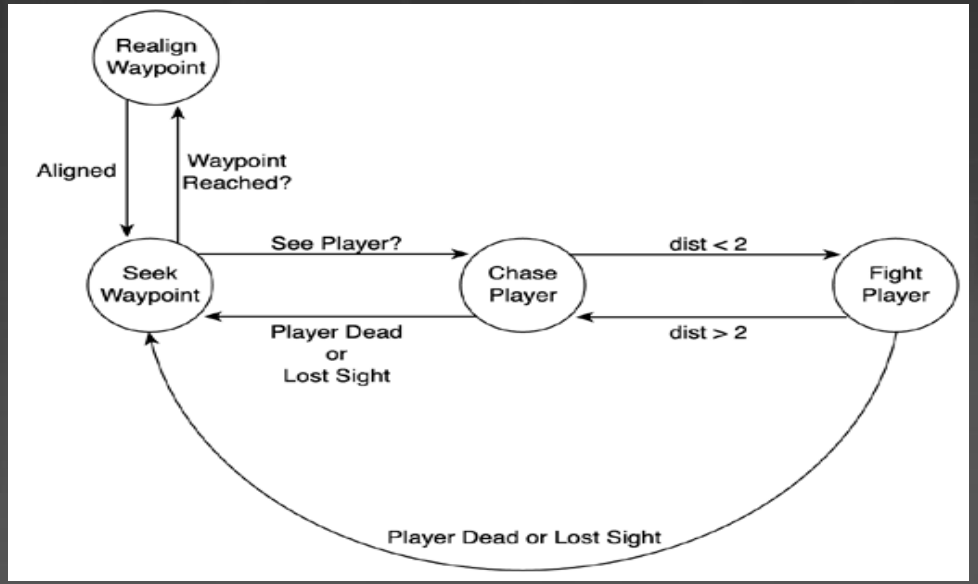
- Sometimes, especially when modeling complex behaviors, a classic FSM will begin to grow quickly, becoming cluttered and unmanageable.
- Even worse, we will sometimes need to add some additional behavior and will discover how our FSM's size almost doubles in each step.
- FSMs are a great tool, but scaling them up is not always a trivial task. Thus, some extensions to this model exist that allow greater control over complex AI systems.
- Using parallel automata is one of the most popular approaches because it allows us to increase the complexity while limiting the size of the automata.

Game AI Techniques-Parallel Automata

- The core idea of parallel automata is to divide our complex behavior into different subsystems or layers, pretending that the entity being modeled has several brains.
- By doing so, each sub-brain is assigned a simpler automata, and thus the overall structure is easier to design and understand.
- In the previous example let’s say that the character can even shoot you.

Parallel Automata

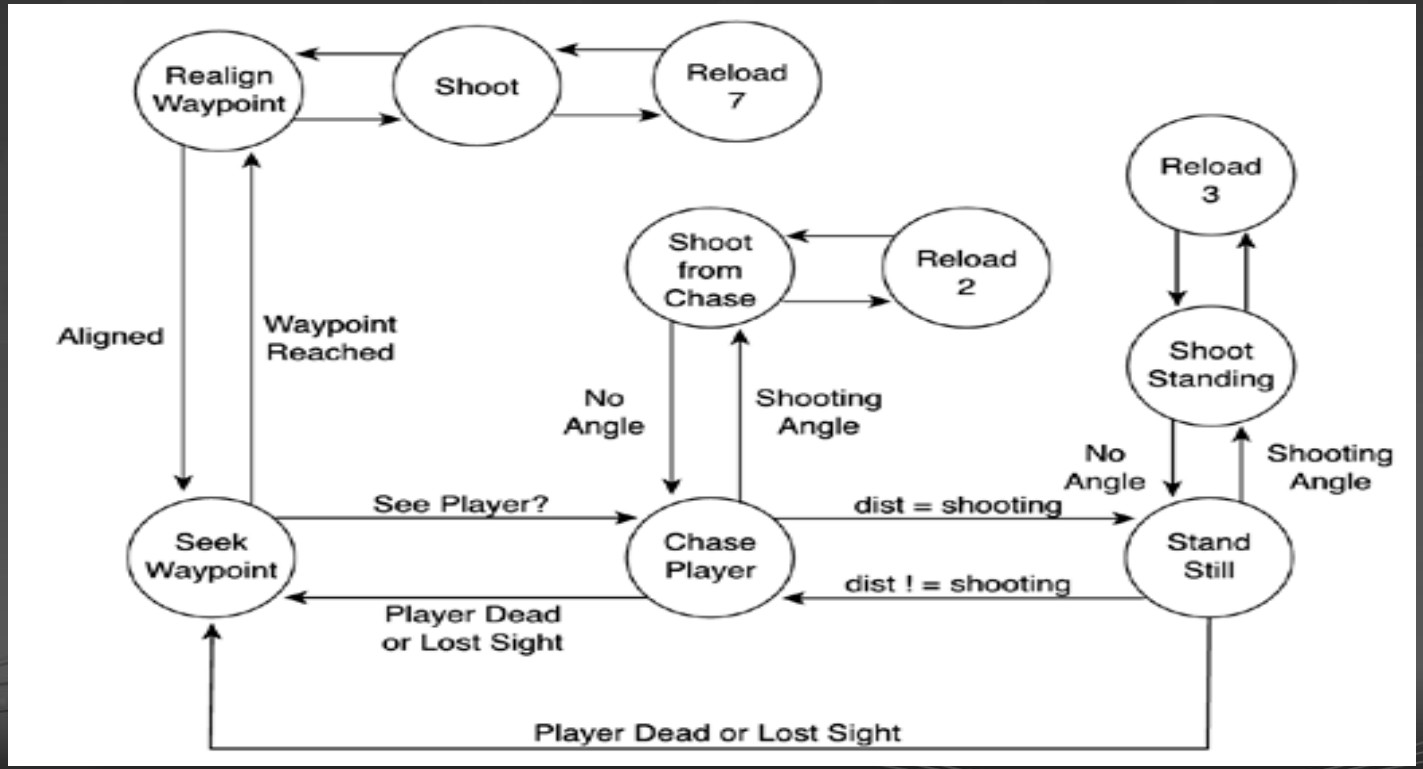




AI

Parallel Automata

This is what happens if we didn't use Parallel Automata



Synchronized FSM

- Another addition to our FSM bag of tricks is implementing inter-automata communications, so several of our AIs can work together and, to a certain extent, cooperate.
- By doing so we can have enemies that attack in squads. While one of them advances, the other provides coverage, and the third one calls for help.
- Such an approach became very popular with the game Half-Life, which was considered a quantum leap in action AI, basically because of the team-based AI.

Synchronized FSM

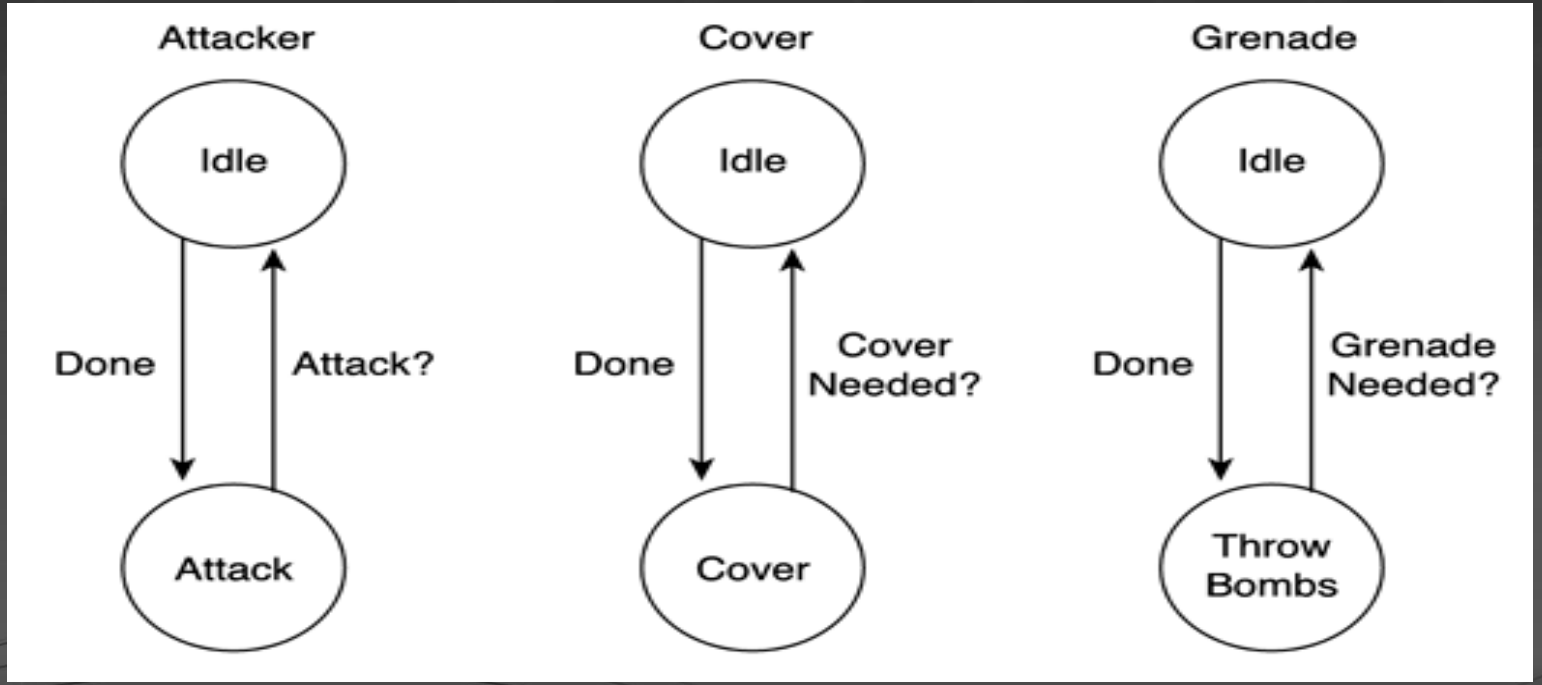
- At the core, synchronizing AIs involves a shared memory area where all automata can write to and read data from. Like a bulletin board metaphor.
- Whenever an AI wants to say something to the rest, it posts it in the shared memory area, so others can browse the information; and if it is relevant to them, they take it into consideration in their decision-making process.
- There are two common approaches to this task.
  - One is to use a message passing architecture, which involves sending messages to those AIs we want to synchronize to.
  - If an AI needs to keep in sync with several others, many messages will need to be sent.
  - The alternative and preferred method is to use a bulletin board architecture, where we don't send dedicated messages, but instead post our sync messages on a shared space that is checked frequently by all AIs.

Synchronized FSM

- Let's make a synchronized AI for a squad of three soldiers, designed to attack an enemy in a coordinated manner.
- I will override their navigation behavior and focus on the actual combat operation.
  - As one of the three establishes contact with the player, he will designate himself as the attacker.
  - The role of the attacker AI is to wait until a second AI, called the cover, begins providing cover fire.
  - Then, the attacker will take advantage of the cover fire to advance toward the player and try to shoot him down.
  - A third AI, dubbed the grenade, will remain in the back behind some protection, throwing grenades.
  - If any soldier gets killed, the remaining soldiers will shift in the chain of command to cover the position.

AI

Synchronized FSM



Synchronized FSM

- we will implement the synchronization mechanism using a shared memory that will store the following data:
  - bool fighting; // set by the first automata that contacts the enemy
  - bool attacker_alive; // true if attacker is alive
  - bool cover_alive; // true if cover is alive
  - bool grenade_alive; // true if grenade is alive

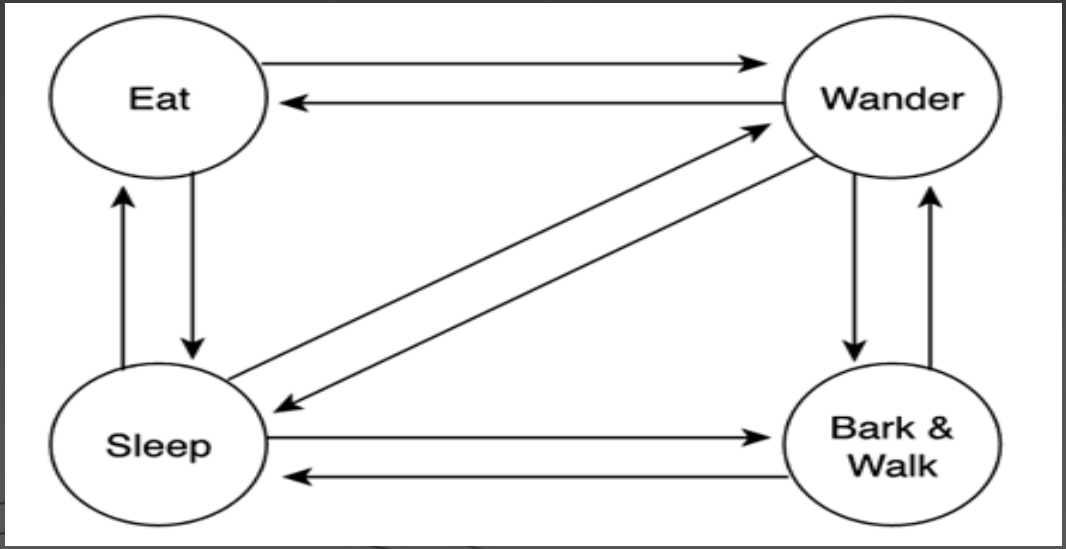
- Our state machine code will need two operations: a shared memory read, which will always be part of the transition test, and a shared memory write, which will be performed from a state's own code. For example, the code used to detect the enemy would look something like this:
- if (enemy sighted)
  - if (fighting=false)
    - fighting=true
    - attacker_alive=true;

Rule Systems

- Finite state machines are a very convenient tool for designing AIs. They can model different behaviors elegantly. But some phenomena are not easy to describe in terms of states and transitions.
- For example, imagine the following specs for a virtual dog:
  - If there's a bone nearby and I'm hungry, I'll eat it.
  - If I'm hungry (but there is no bone around), I will wander.
  - If I'm not hungry, but I'm sleepy, I will sleep.
  - If I'm not hungry and not sleepy, I'll bark and walk.
- What happens if we use FSMs for this ?

Rule Systems

- We will end up with a FSM like below:



Rule Systems

- The key idea is that FSMs are well suited for behaviors that are:
  - Local in nature (while we are in a certain state, only a few outcomes are possible).
  - Sequential in nature (we carry out tasks after other tasks depending on certain conditions
- The virtual dog just described is not local. If you analyze it, all states can yield any other state, so the model is not local at all.
- Also, there are no visible sequences. All the dog actually does is act according to some priorities or rules. Luckily, there is one way to model this kind of prioritized, global behavior.

Rule Systems

- At the core of an RS, there is a set of rules that drive our AI's behavior. Each rule has the form:
- Condition -> Action
- In the previous dog example, a more formal specification of the system would be
  - (Hungry) & (Bone nearby) -> Eat it
  - (Hungry) & (No bone nearby) -> Wander
  - If (not hungry) & (Sleepy) -> Sleep
  - If (not hungry) & (Not sleepy) -> Bark and walk

Rule Systems

- RSs, as opposed to FSMs, provide a global model of behavior.
- At each execution cycle, all rules are tested, so there are no implied sequences.
- This makes them better suited for some AIs.
- Specifically, RSs provide a better tool when we need to model behavior that is based on guidelines.
- We model directions as rules, placing the more important ones closer to the top of the list, so they are priority executed. Imagine that we need to create the AI for a soldier in a large squadron. The rule system could be
  - If in contact with an enemy -> combat
  - If an enemy is closer than 10 meters and I'm stronger than him -> chase him
  - If an enemy is closer than 10 meters -> escape him
  - If we have a command from our leader pending -> execute it
  - If a friendly soldier is fighting and I have a ranged weapon -> shoot at the enemy
  - Stay still

Tactical AI

Tactical AI

- Lets just cover topics about path finding
- But we will come back and delve deeper

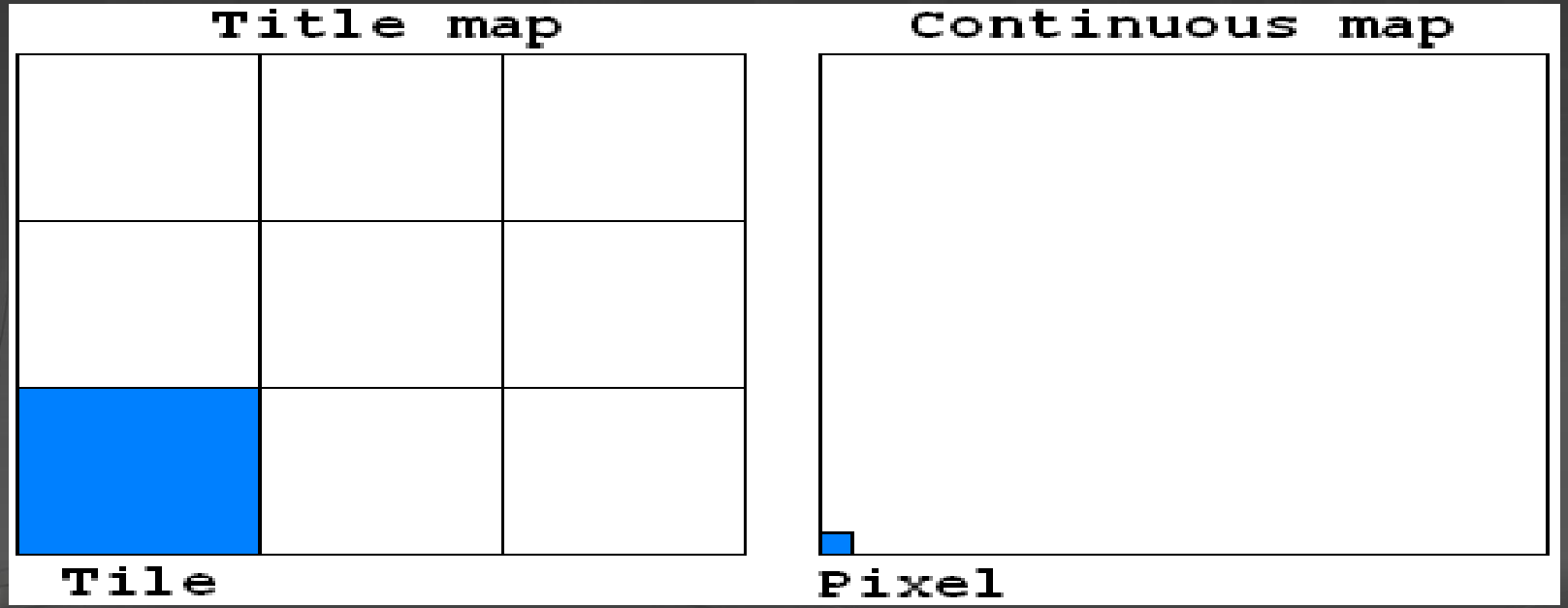
Path Finding

- Path finding on Gridded Maps
  - Breath First Search
  - Depth First Search
  - Best First Search
  - And the most common and dominant one
    - A*
  - Path Finding on Non- Gridded Maps
    - More on them later

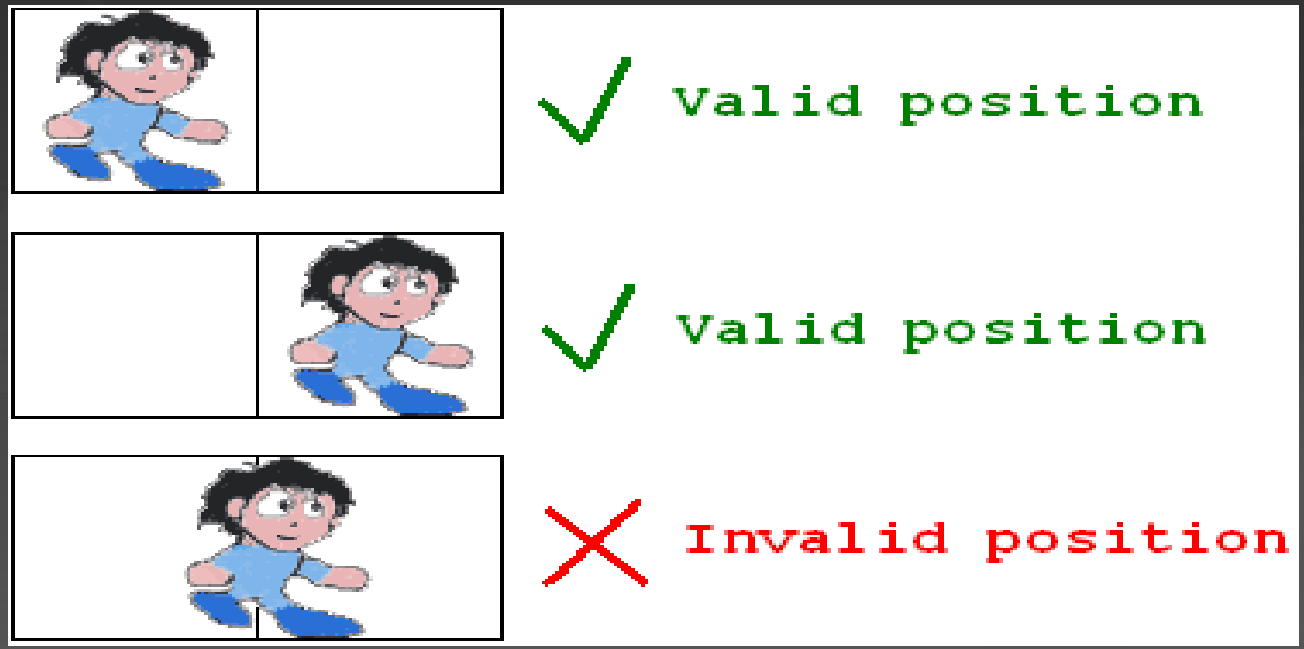
A* Path finding for games

Tile Maps

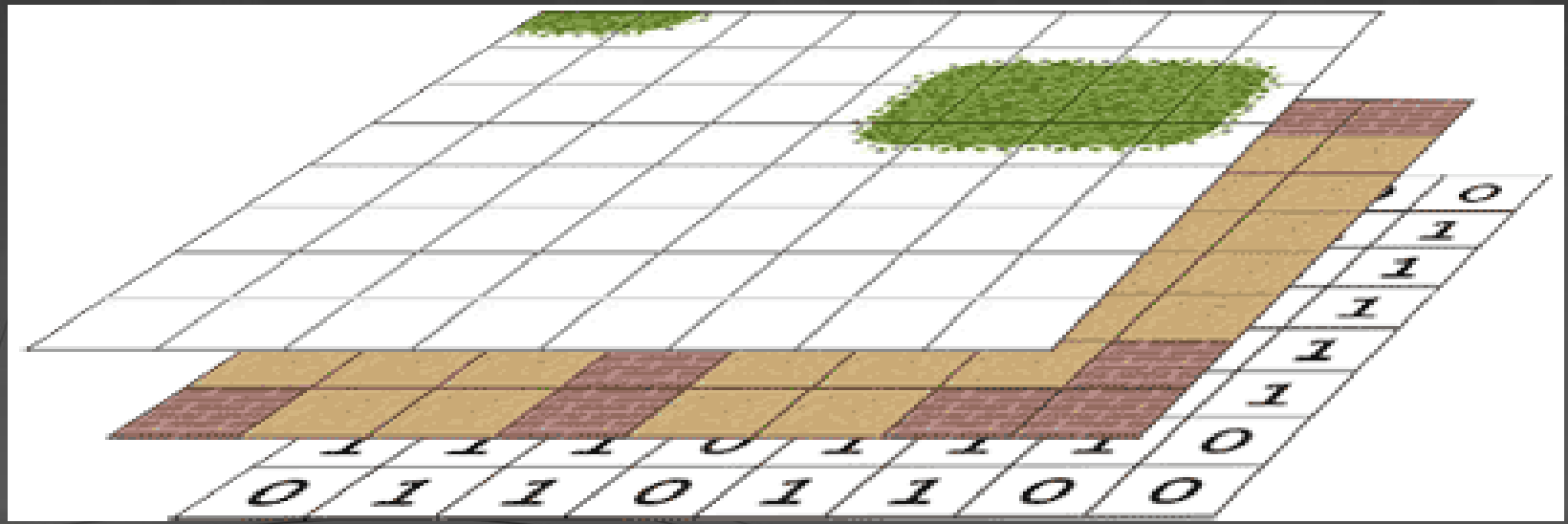
- We have two kinds of maps: continues maps and tile maps
- The building block in continues maps is **Pixel**
- The building block in tile maps is **tiles**



- In tile maps you can only walk on tiles and you move per tiles



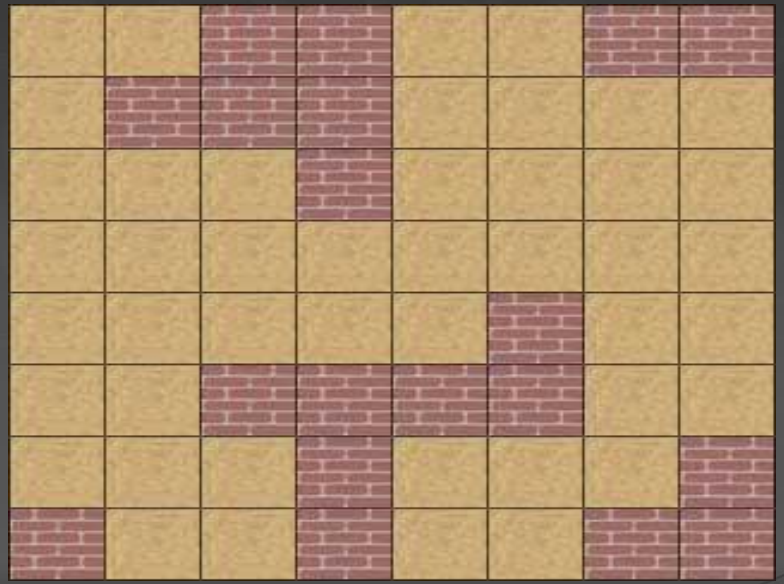
- So in order to know which tiles you can walk in and which tiles you can’t you have to have a layer below any graphical layers that marks the tiles that you can walk in.
- This layer is called the WALKABLE LAYER



AI

- The only thing that you should care about is the walkable layer
- You can use 2d arrays or XML files to construct the walkable layer

1	1	0	0	1	1	0	0
1	0	0	0	1	1	1	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1
1	1	0	0	0	0	1	1
1	1	1	0	1	1	1	0
0	1	1	0	1	1	0	0



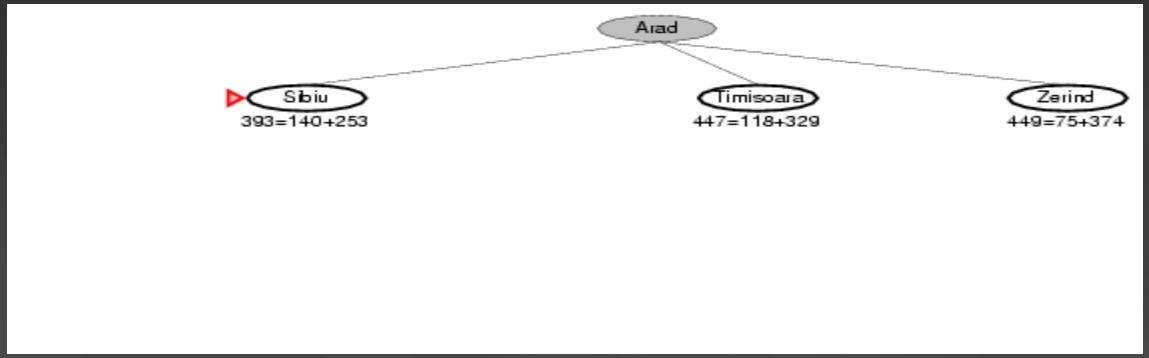
A really short introduction to A*

- A* is an informed search and that means that you have some guess or some heuristic information about how much it's going to cost you to get to the goal

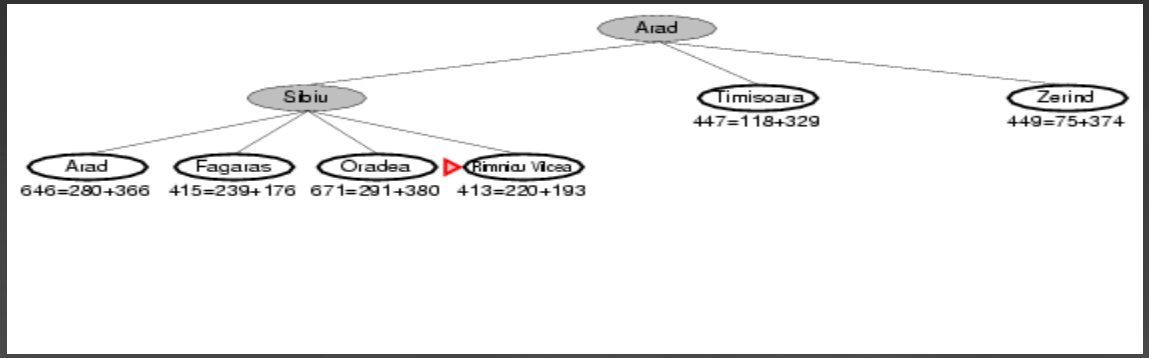
- The golden formula:
  - F(n)=g(n)+h(n)
    - g(n) is the path cost from start node to node n
    - h(n) is the estimated cost of the cheapest path from node n to the goal
  - H(n) is called the heuristic function
  - If H(n) is admissible then A* is both complete and optimal
    - admissible is that h(n) does not overestimate the cost to get to the goal

A* search example

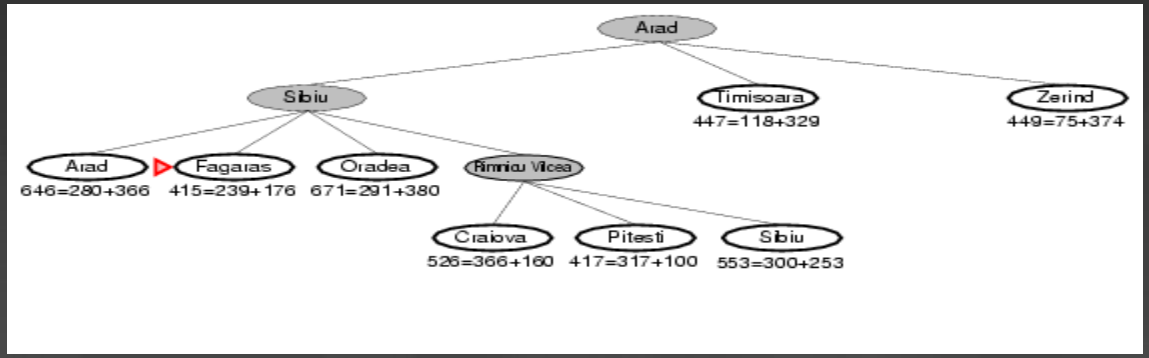
A* search example



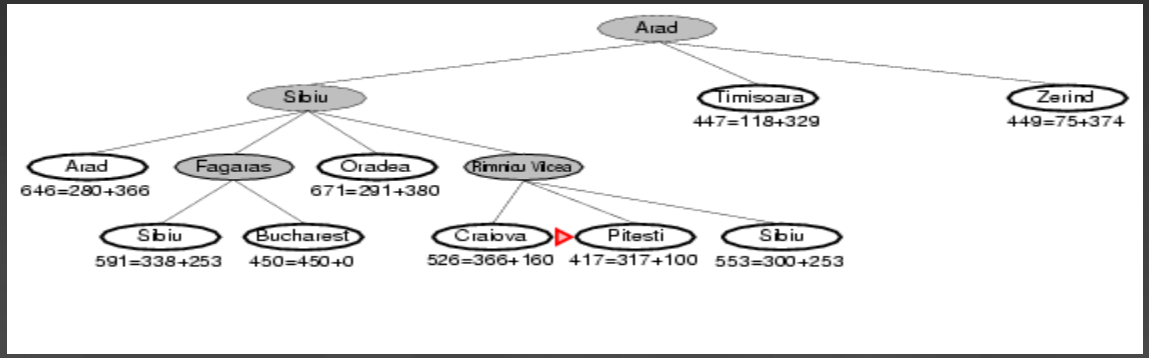
A* search example



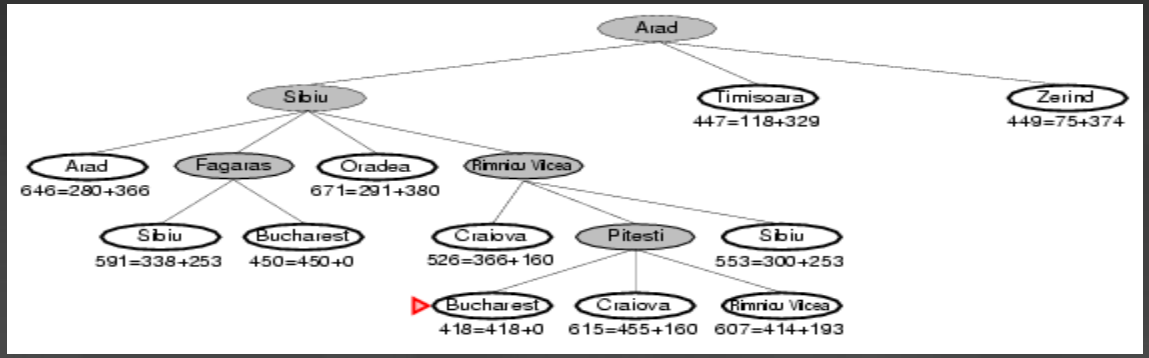
A* search example



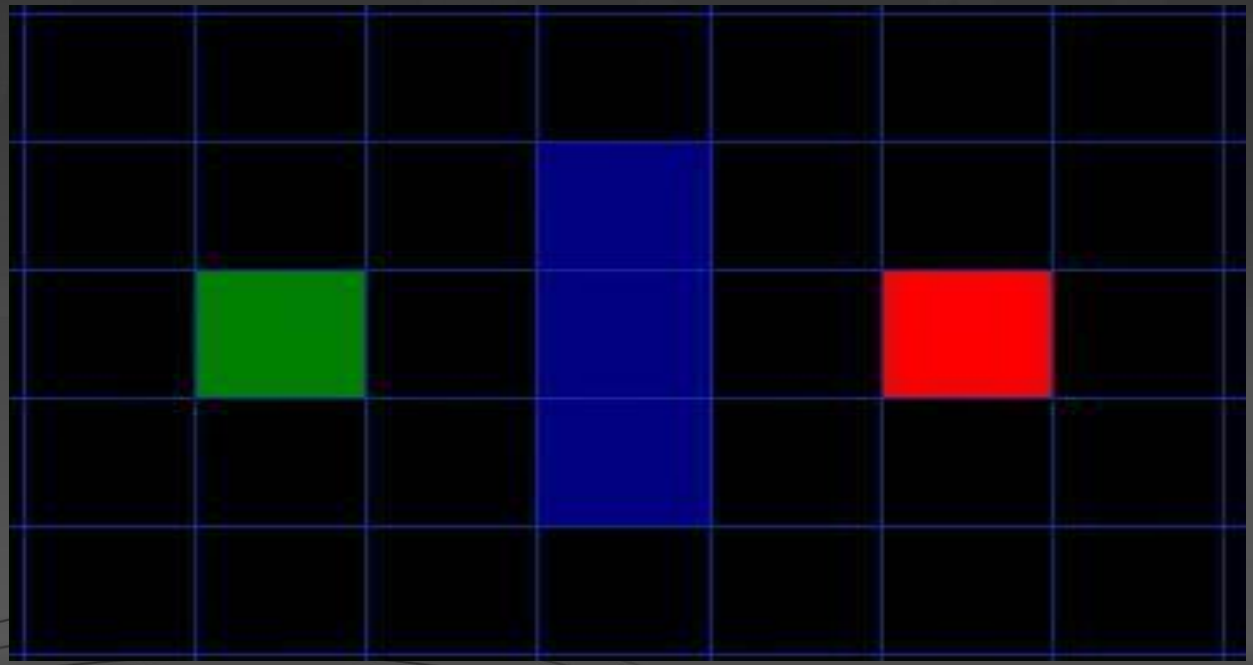
A* search example



A* search example



- Let’s Assume that we have an environment like the picture below and we want to get from the starting node(Green tile) to the target (Red tile) and there is a wall in between



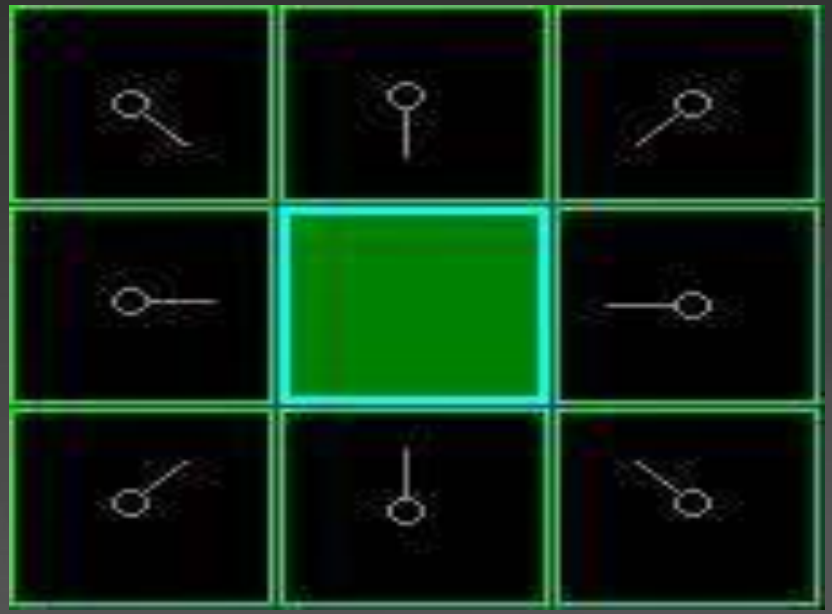
- We begin the search by doing the following:

- Begin at the starting point A and add it to an “open list” of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.

- Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its “parent square”. This parent square stuff is important when we want to trace our path.

- Drop the starting square A from your open list, and add it to a “closed list” of squares that you don’t need to look at again for now.

- At this point, you should have something like the following illustration



- Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

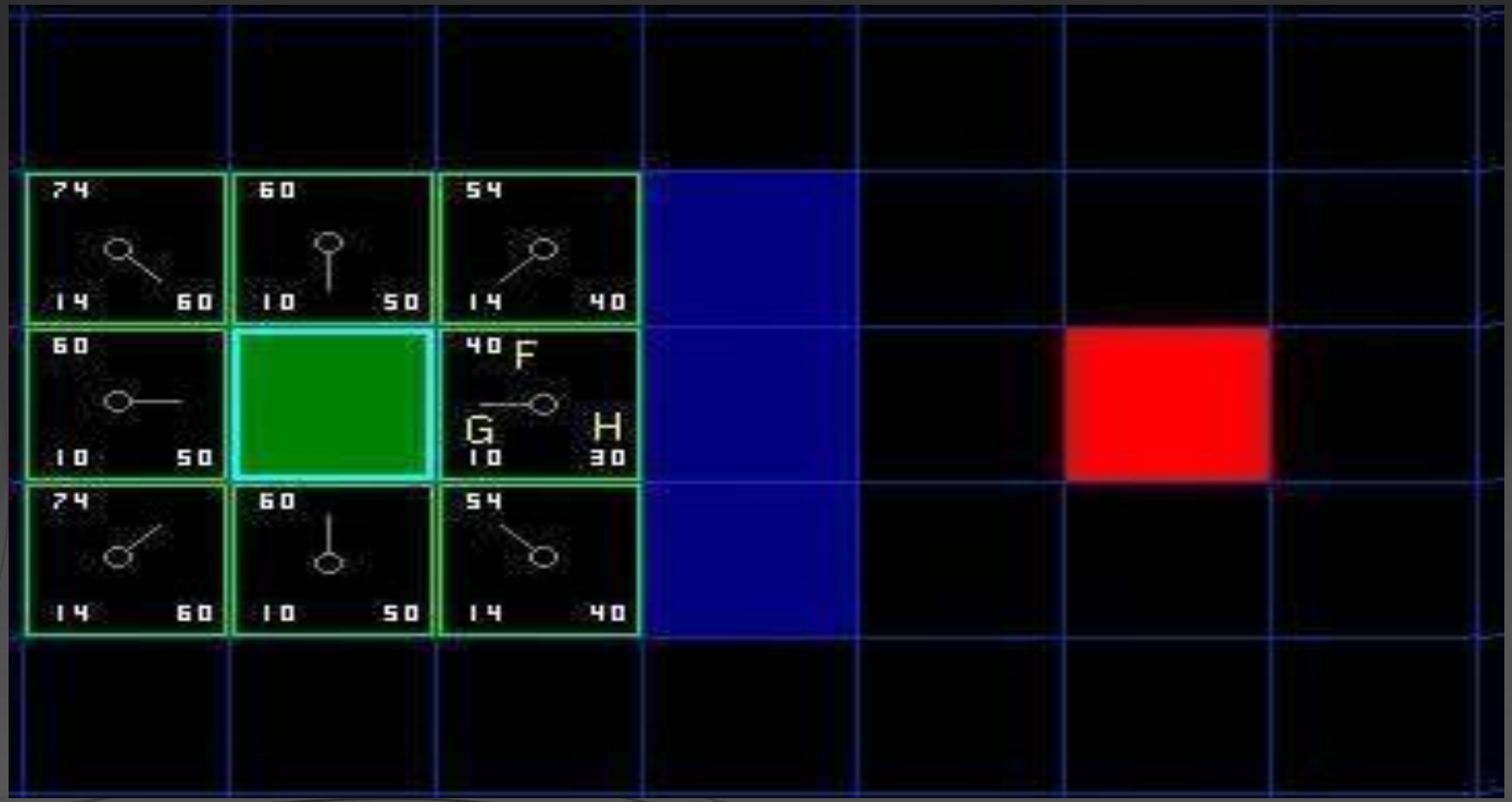
- So In order to calculate F cost you have to calculate G and H

- In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move.
- We use these numbers because the actual distance to move diagonally is the square root of 2 or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity's sake.
- The ratio is about right, and we avoid having to calculate square roots and we avoid decimals

- So how do we choose the H?

- H can be estimated in a variety of ways.
- The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square
- ignoring diagonal movement, and ignoring any obstacles that may be in the way.

- The result of the first step is:



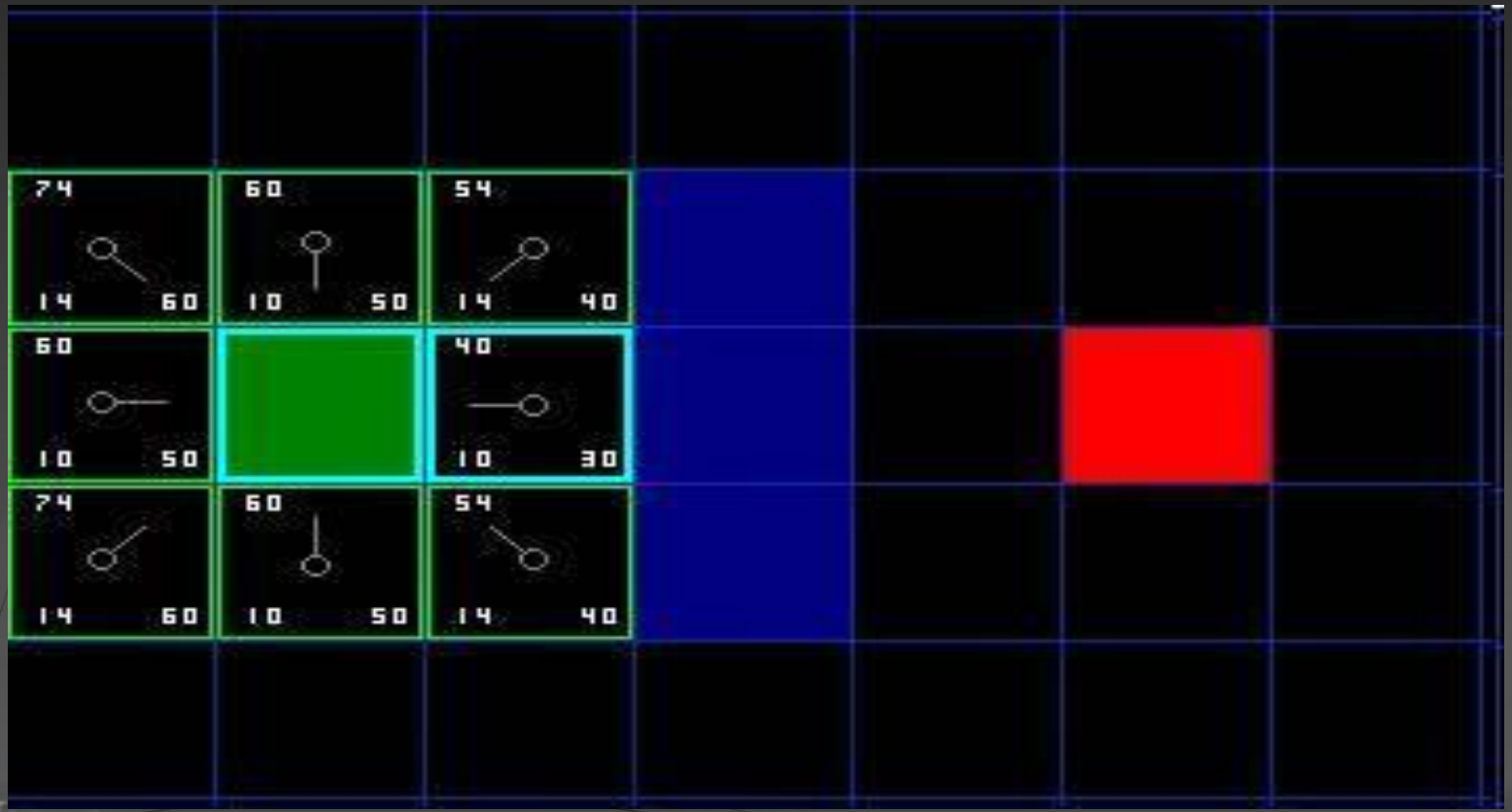
- **Continuing the Search**

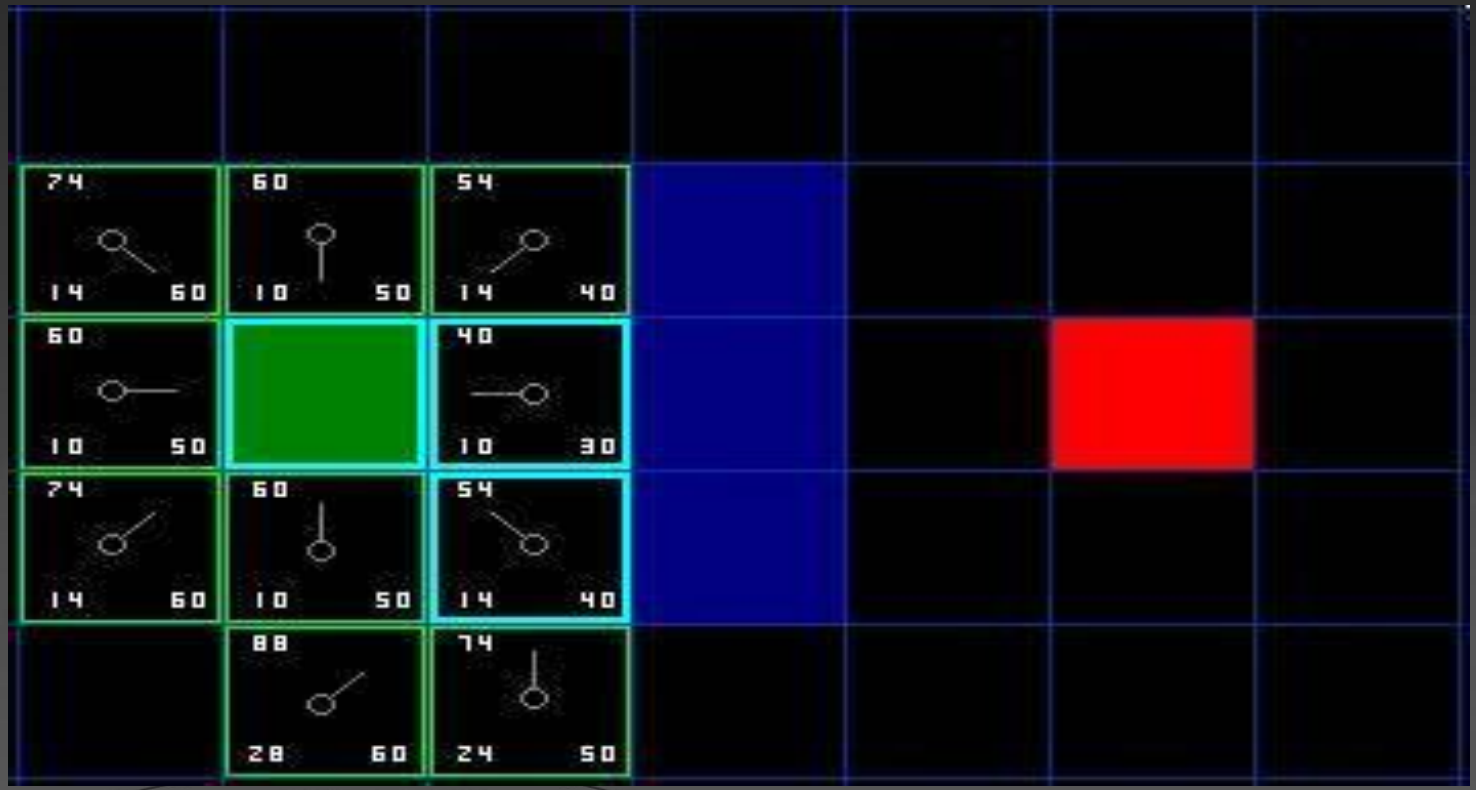
- To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square: Drop it from the open list and add it to the closed list.

- Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the "parent" of the new squares.

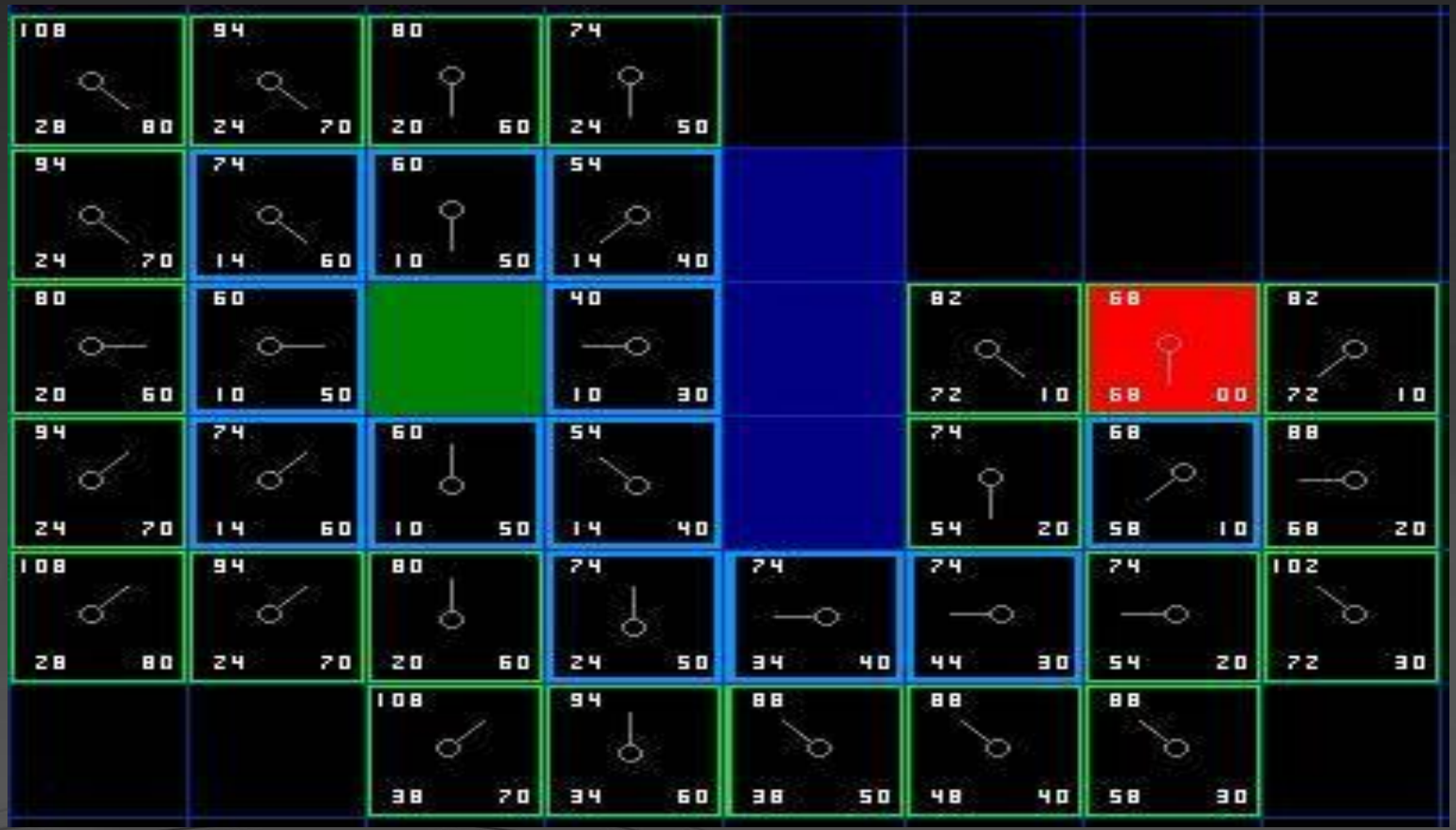
- If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don't do anything.
  On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square).

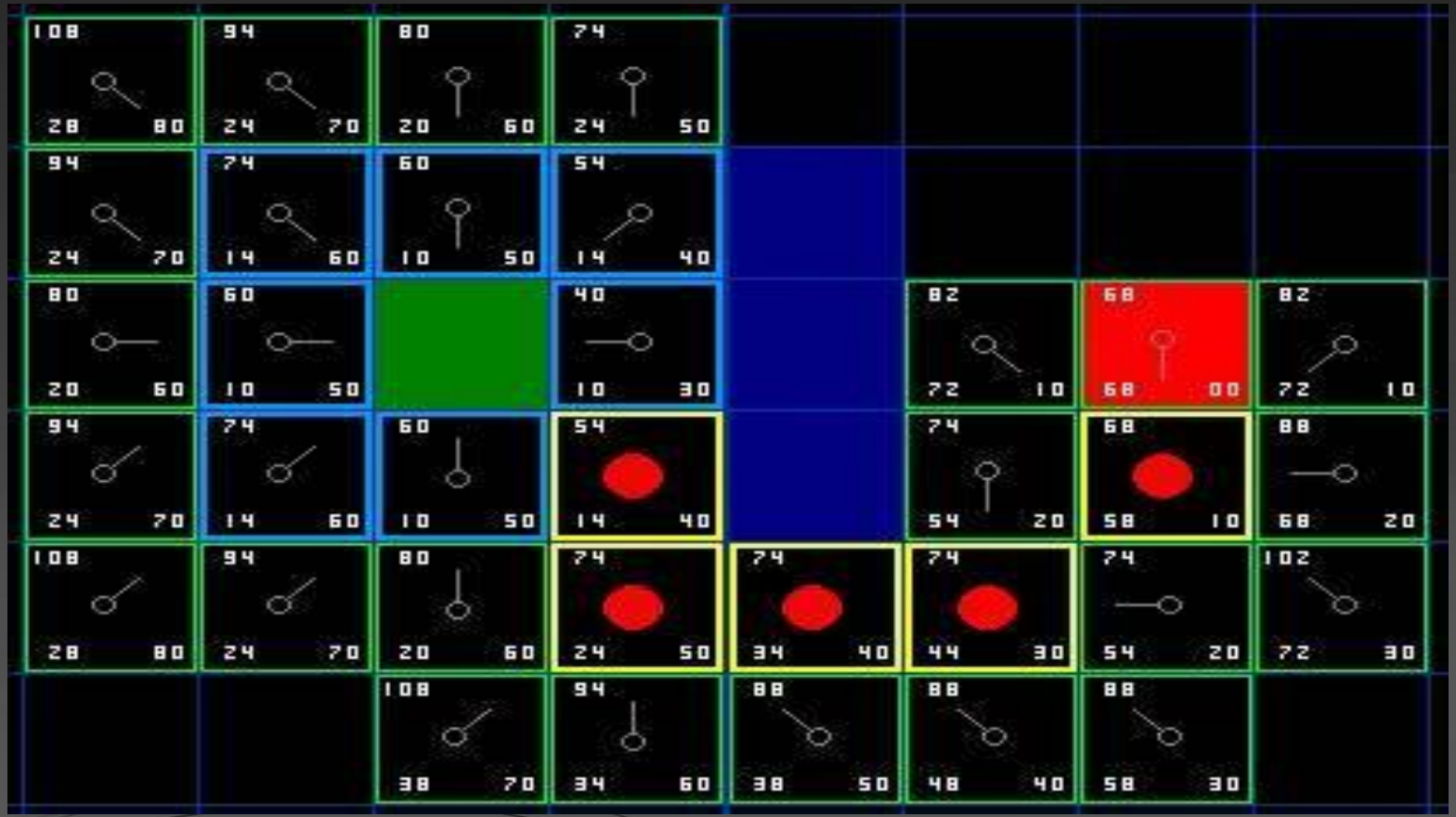
- So we choose the tile with the lowest F cost




74
14
60
60
10
50
54
14
40
60
10
50
40
10
30
74
14
60
60
10
50
54
14
40
88
28
60
74
24
50

AI


108 28 80
94 24 70
80 20 60
74 24 50
94 24 70
74 14 60
60 10 50
54 14 40
80 20 60
60 10 50
40 10 30
82 72 10
68 68 00
82 72 10
94 24 70
74 14 60
60 10 50
54 14 40
74 54 20
68 58 10
88 68 20
108 28 80
94 24 70
80 20 60
74 24 50
74 34 40
74 44 30
74 54 20
102 72 30
108 38 70
94 34 60
88 38 50
88 48 40
88 58 30


108 28 80
94 24 70
80 20 60
74 24 50
94 24 70
74 14 60
60 10 50
54 14 40
80 20 60
60 10 50
40 10 30
82 72 10
68 68 00
82 72 10
94 24 70
74 14 60
60 10 50
54 14 40
74 54 20
68 58 10
88 68 20
108 28 80
94 24 70
80 20 60
74 24 50
74 34 40
74 44 30
74 54 20
102 72 30
108 38 70
94 34 60
88 38 50
88 48 40
88 58 30

Final Algorithm

- 1) Add the starting square (or node) to the open list.
- 2) Repeat the following:
  - a) Look for the lowest F cost square on the open list. We refer to this as the current square.
  - b) Switch it to the closed list.
  - c) For each of the 8 squares adjacent to this current square …
    - If it is not walk able or if it is on the closed list, ignore it. Otherwise do the following.
    - If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
    - If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.
  - d) Stop when you:
- Add the target square to the closed list, in which case the path has been found or
- Fail to find the target square, and the open list is empty. In this case, there is no path.
- 3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

AI