## Finite State Machines

Finite state machines can have 3 states for the Pacman controller. These 3 states would be

1. Seeking pellets
2. Evading ghosts
3. Evade ghost
4. A fourth one could be something that Kevin suggested where pacman roams around the power pill instead of eating it, but transition phase will have to be designed

Pros:

1. The design is simple, easy to implement and understand

Cons

1. they are not flexible and dynamic (a criticism of all ad-hoc behaviour authoring methods)
2. after their design is completed, tested and debugged there is limited room for adaptation and evolution

a pathfinding algo can be used to maximize the number of pellets eaten

Tree Search

Could be a better implementation than fsm as they are more flexible to design. One of the nodes which is move is represented as follows:

Diagram

Description automatically generated

More info can be found on week 4 lecture slide

We can again implement the state 4 I have metioned in FSM, this time under move node.

We can use A\* that is already implemented as a path finding algo. If someone has any changes they can do to make A\* better, we can look into it. Things like optimising the cost

Evolutionary Computation

Sample grammar

< grammar > ::= <sel - stat >

<sel - stat > ::= if (\_ < cond > \_ ){ \_ < stat > \_ } \_else {\_ < stat > \_ } | if (\_ < cond > \_ ){ \_ < stat > \_ }

< stat > ::= < action > | <sel - stat >

< action > ::= escape | attack | seekFood

< cond > ::= <num - st >\_ < num - op >\_ < num >

<num - st > ::= dist\_closest\_NE\_ghost | dist\_closest\_E\_ghost

<num - op > ::= EQ | NE | LT | GT | LE | GE

< numb > ::= 0 | 5 | 10 | ... | 40

Or

< gram > ::= <sel - stat >

<sel - stat > ::= if (\_ < cond > \_ ){ \_ < stat > \_ } \_else {\_ < stat > \_ } | if (\_ < cond > \_ ){ \_ < stat > \_ }

< stat > ::= < action > | <sel - stat >

< action > ::= run\_to\_closest\_pill | run\_to\_closest\_ppill | run\_to\_closest\_E\_ghost | run\_from\_closest\_NE\_ghost

< cond > ::= < bool - st > | <num - st >\_ < num - op >\_ < num >

< bool - st > ::= < bool - api > | not \_ < bool - api >

< bool - api > ::= is\_junction

<num - st > ::= dist\_closest\_NE\_ghost | dist\_closest\_NE\_ghost | dist\_closest\_pill | ...

<num - op > ::= EQ | NE | LT | GT | LE | GE

< numb > ::= 0 | 5 | 10 | ... | 40

Reference <http://ceur-ws.org/Vol-1957/CoSeCiVi17_paper_12.pdf>

Reinforcement Learning

Using q learning

Reference <http://ceur-ws.org/Vol-1957/CoSeCiVi17_paper_14.pdf>

This method should give good results but would be hard to implement especially in python, we can maybe break it down in 2 groups with 1 group trying to implement it in java and the other group implementing this in python and trying to link it to the java controller. Both of these approaches seem like a waste of time.

## Remarks

The first 2 algos and the ideas are directly taken from James’s slides and most people would be implementing that.

These should be easier to implement and things like A\* are already there. Implementing GE or reinforcement would be hard for us especially because we would have to code it from scratch in JAVA. We should probably go for FSM or Tree Search, whichever can give better results.

## Note

This document is there just to discuss the ideas and not as the final thing. We have to choose one of these and do a detailed analysis on why we chose the particular method while comparing it to others.