Game AI

Project 2: Game Trees and Path Planning

Alina Arunova
Phil Matyash
Marina Mircheska
Aditya Kela
Sattar Rahimbeyli

Outline

- Tic Tac Toe tree calucaltion
- Minmax computations
- Minmax search for *connect four*
- Breakout predictor
- Path planning

Task 2.1: Tic Tac Toe tree calculation

Task: Build a full tic tac toe tree. Calculate wins by X player, branching factor.
Solution:

1) to generate a tree like structure, the best way is to operate “node” objects with generic methods like *setChild, getChild, hasChild, getParent, setParent* and application specific *setState, getState*

2) recursively build the tree

3) recursively navigating by tree

- calculate the properties

Task 2.1: Tic Tac Toe tree calculation

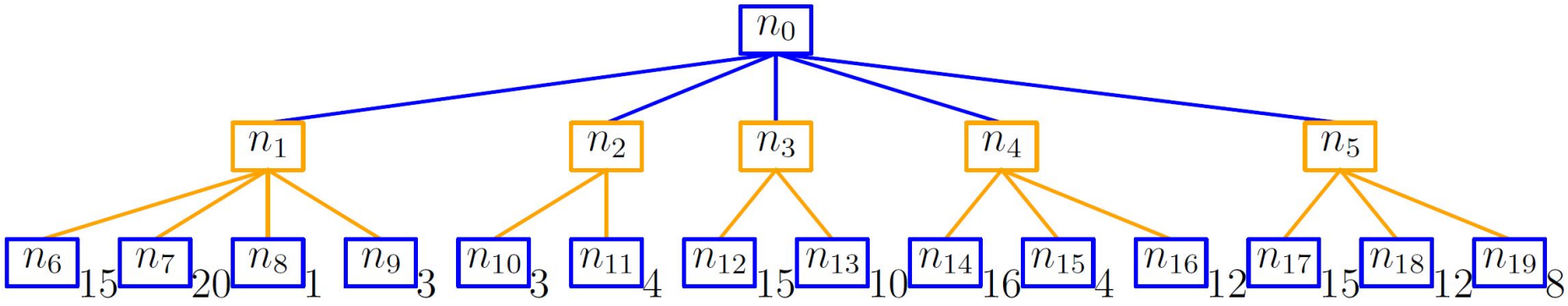
- The results:
- Upper bound of nodes = 9! (362880)
-
- Total node count :269173
- Total win X player win count :55872
- Total draw count :95166
- Total parent node count :155758
- Average branching factor: 1.728149

Task 2.2: Minmax computations

Task 2a: implement the *minmax* algorithm and test it on an example tree

Minmax algorithm consists of recursive computations:

$$mmv(n) = \begin{cases} u(n) & \text{if } n \text{ is a terminal node} \\ \max\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MAX \text{ node} \\ \min\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MIN \text{ node} \end{cases}$$

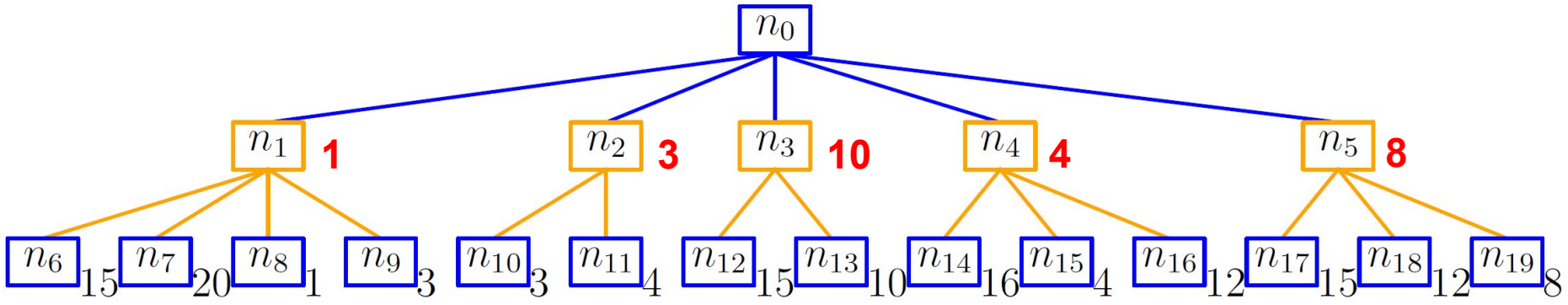


Task 2.2: Minmax computations

Task: implement the *minmax* algorithm and test it on an example tree

Minmax algorithm consists of recursive computations:

$$mmv(n) = \begin{cases} u(n) & \text{if } n \text{ is a terminal node} \\ \max\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MAX \text{ node} \\ \min\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MIN \text{ node} \end{cases}$$

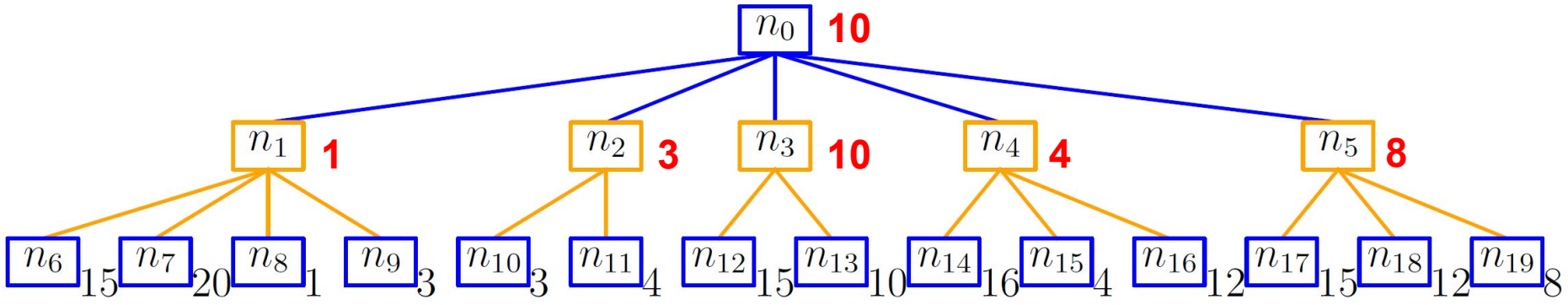


Task 2.2: Minmax computations

Task: implement the *minmax* algorithm and test it on an example tree

Minmax algorithm consists of recursive computations:

$$mmv(n) = \begin{cases} u(n) & \text{if } n \text{ is a terminal node} \\ \max\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MAX \text{ node} \\ \min\limits_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MIN \text{ node} \end{cases}$$



Task 2.2: Minmax computations

Our implementation:

- class TreeNode()

TreeNode(nodeId, utility)
nodeId: int parent: int successors: list mmv: int maxutil: int
insertChildren(node, nodeList) maxStep(node) minStep(node) bestMaxStrategy(node) printTree(node)

Task 2.2: Minmax computations

Our implementation:

- class TreeNode()
- building a tree from an example done manually ☹

TreeNode(nodeId, utility)
nodeId: int parent: int successors: list mmv: int maxutil: int
insertChildren(node, nodeList) maxStep(node) minStep(node) bestMaxStrategy(node) printTree(node)

```
root = TreeNode(0, None)
node1 = TreeNode(1, None)
node2 = TreeNode(2, None)
node3 = TreeNode(3, None)
node4 = TreeNode(4, None)
node5 = TreeNode(5, None)
node6 = TreeNode(6, 15)
node7 = TreeNode(7, 20)
node8 = TreeNode(8, 1)
node9 = TreeNode(9, 3)
node10 = TreeNode(10, 3)
node11 = TreeNode(11, 4)
node12 = TreeNode(12, 15)
node13 = TreeNode(13, 10)
node14 = TreeNode(14, 16)
node15 = TreeNode(15, 4)
node16 = TreeNode(16, 12)
node17 = TreeNode(17, 15)
node18 = TreeNode(18, 12)
node19 = TreeNode(19, 8)

root.insertChildren(root, [node1, node2, node3, node4, node5])
node1.insertChildren(node1, [node6, node7, node8, node9])
node2.insertChildren(node2, [node10, node11])
node3.insertChildren(node3, [node12, node13])
node4.insertChildren(node4, [node14, node15, node16])
node5.insertChildren(node5, [node17, node18, node19])
```

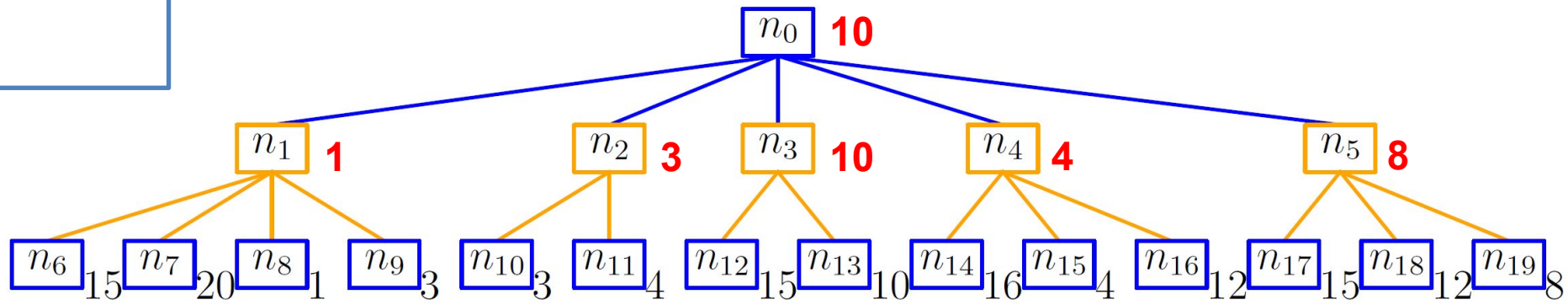
Task 2.2: Minmax computations

Our implementation:

- class TreeNode()
- building a tree from an example done manually ☹
- verbose visualization •

TreeNode(nodeId, utility)
nodeId: int parent: int successors: list mmv: int maxutil: int
insertChildren(node, nodeList) maxStep(node) minStep(node) bestMaxStrategy(node) printTree(node)

```
current node  0
node has children, exploring children...
|nodeId: 1 |parent: 0 |utility: None |mmv: 1 |maxutil: 20
node has children, exploring children...
|nodeId: 6 |parent: 1 |utility: 15 |mmv: 15 |maxutil: -inf
node is a leaf
|nodeId: 7 |parent: 1 |utility: 20 |mmv: 20 |maxutil: -inf
node is a leaf

.............
|nodeId: 5 |parent: 0 |utility: None |mmv: 8 |maxutil: 15
node has children, exploring children...
|nodeId: 17 |parent: 5 |utility: 15 |mmv: 15 |maxutil: -inf
node is a leaf
|nodeId: 18 |parent: 5 |utility: 12 |mmv: 12 |maxutil: -inf
node is a leaf
|nodeId: 19 |parent: 5 |utility: 8 |mmv: 8 |maxutil: -inf
node is a leaf

mmv for n0:  10
```



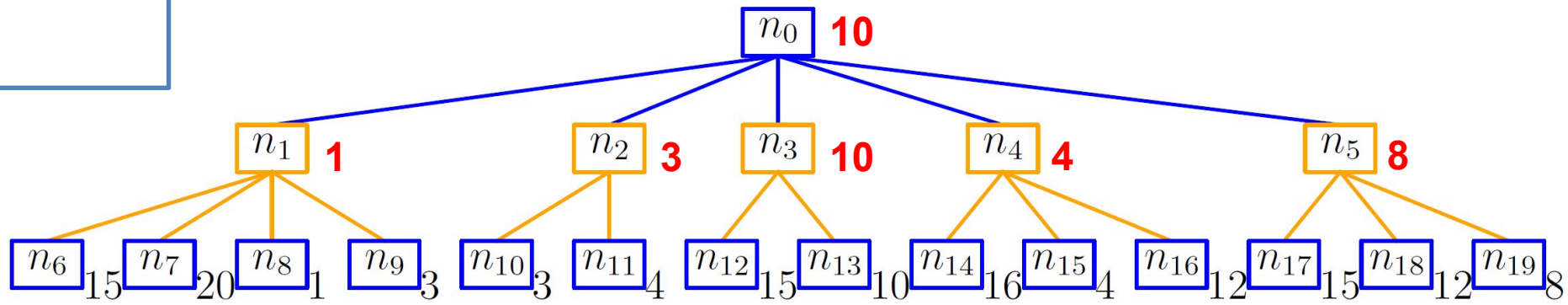
Task 2.2: Minmax computations

Our implementation:

- class TreeNode()
- building a tree from an example done manually ☹
- verbose visualization •
- but tree traversals and minmax recursion working great! ☺

TreeNode(nodeId, utility)
nodeId: int parent: int successors: list mmv: int maxutil: int
insertChildren(node, nodeList) maxStep(node) minStep(node) bestMaxStrategy(node) printTree(node)

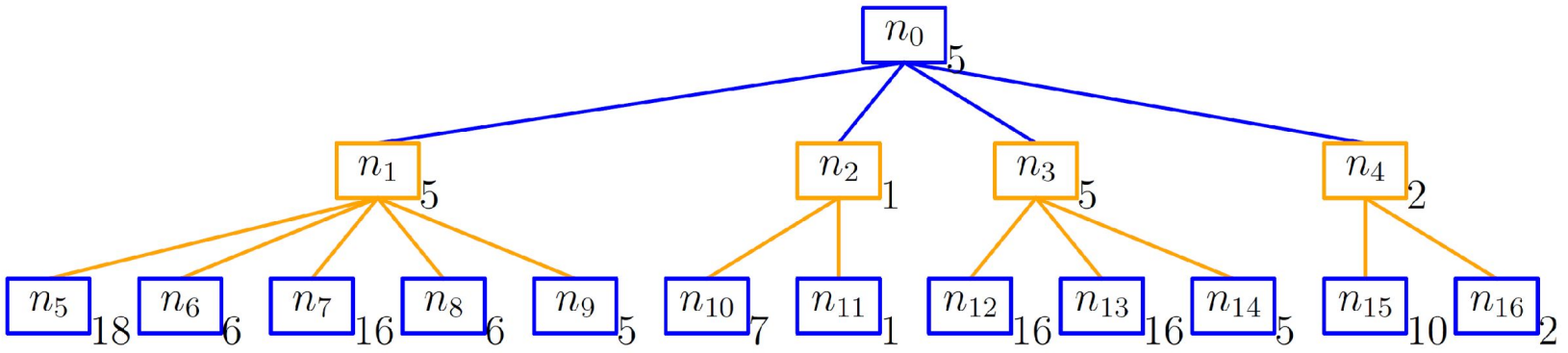
```
current node  0
node has children, exploring children...
|nodeId: 1 |parent: 0 |utility: None |mmv: 1 |maxutil: 20
node has children, exploring children...
|nodeId: 6 |parent: 1 |utility: 15 |mmv: 15 |maxutil: -inf
node is a leaf
|nodeId: 7 |parent: 1 |utility: 20 |mmv: 20 |maxutil: -inf
node is a leaf

.............
|nodeId: 5 |parent: 0 |utility: None |mmv: 8 |maxutil: 15
node has children, exploring children...
|nodeId: 17 |parent: 5 |utility: 15 |mmv: 15 |maxutil: -inf
node is a leaf
|nodeId: 18 |parent: 5 |utility: 12 |mmv: 12 |maxutil: -inf
node is a leaf
|nodeId: 19 |parent: 5 |utility: 8 |mmv: 8 |maxutil: -inf
node is a leaf

mmv for n0:  10
```



Task 2.2: Minmax computations

Task 2b: given a tree with precomputed minmax values, discuss ***ties***

- **Q1:** how to track “better alternatives” in case of ties?

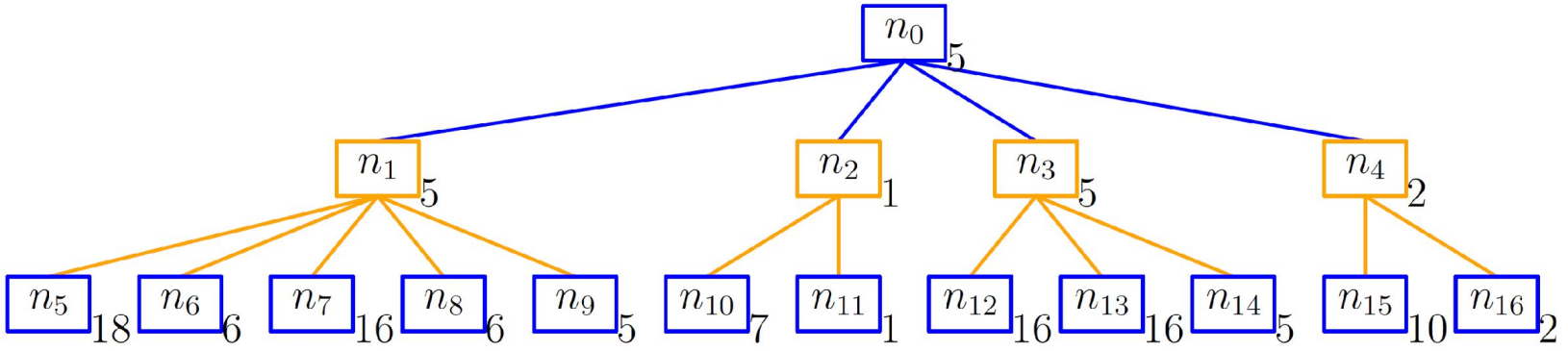


Task 2.2: Minmax computations

Task 2b: given a tree with precomputed minmax values, discuss ***ties***

- **Q1:** how to track “better alternatives” in case of ties?

- **Q2**: does it matter which choice?



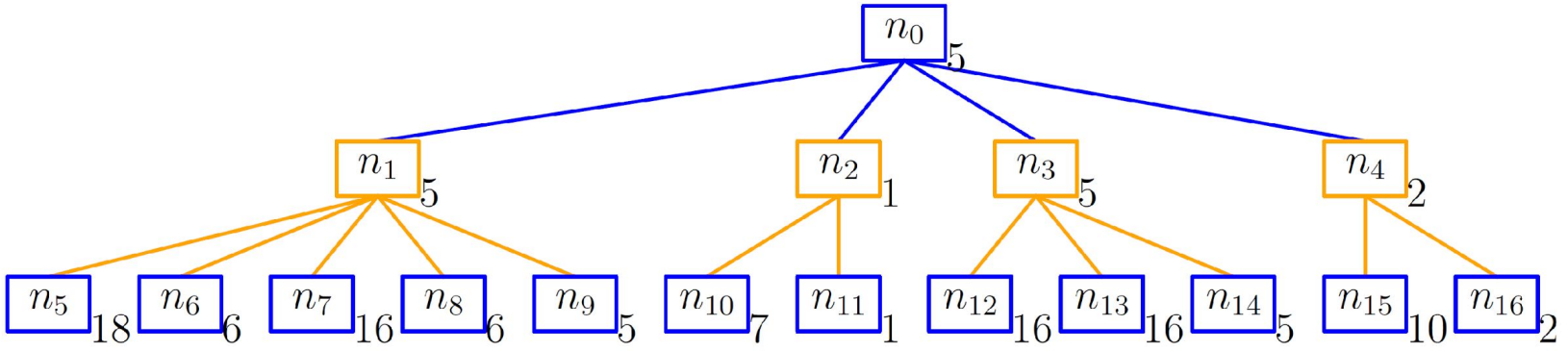
Task 2.2: Minmax computations

Task 2b: given a tree with precomputed minmax values, discuss ***ties***

- **Q1:** how to track "better alternatives" in case of ties?
  - add an additional field (maxutil) for each node
  - while traversing the tree to compute *mmv*, also compute the "best" alternative and store it in each non-leaf node

- **Q2**: does it matter which choice?

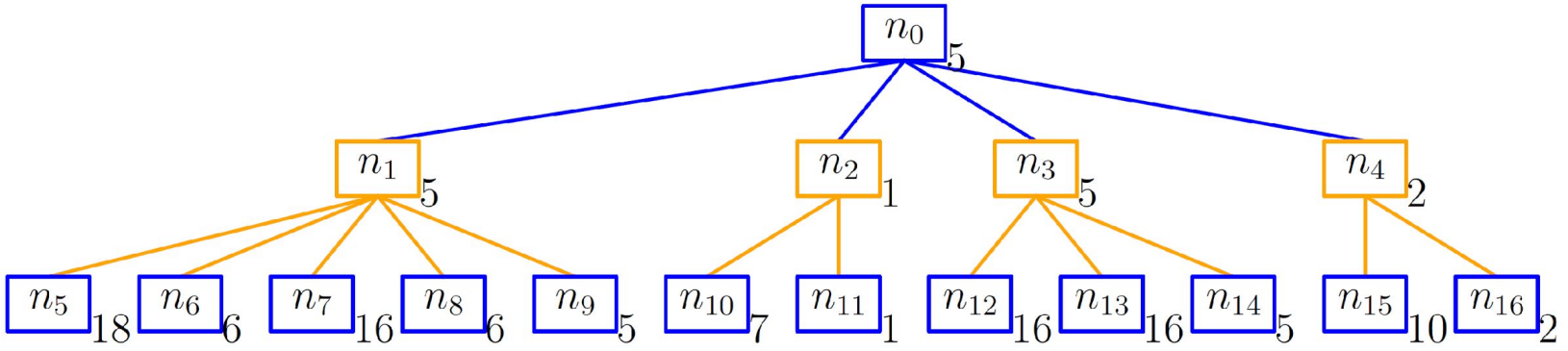
TreeNode(nodeId, utility)
nodeId: int parent: int successors: list mmv: int maxutil: int
insertChildren(node, nodeList) maxStep(node) minStep(node) bestMaxStrategy(node) printTree(node)



Task 2.2: Minmax computations

Task 2b: given a tree with precomputed minmax values, discuss ***ties***

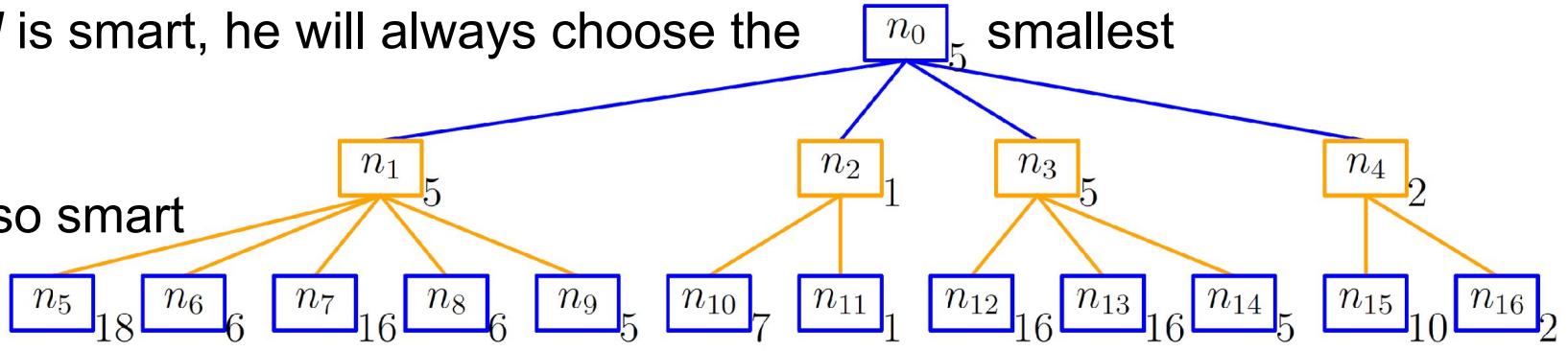
- **Q1:** how to track “better alternatives” in case of ties?
  - add an additional field (maxutil) for each node
  - while traversing the tree to compute *mmv*, also compute the “best” alternative and store it in each non-leaf node
  - when looking for the best path, compare by *maxutil* when ties for *mmv*
- **Q2**: does it matter which choice?



Task 2.2: Minmax computations

Task 2b: given a tree with precomputed minmax values, discuss ***ties***

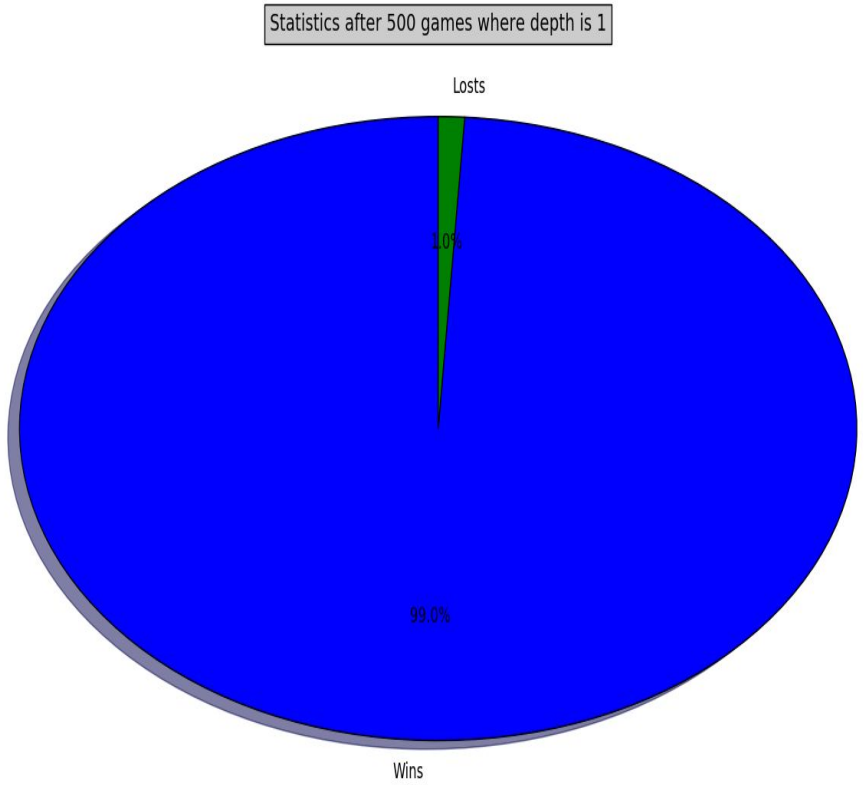
- **Q1:** how to track "better alternatives" in case of ties?
  - add an additional field (maxutil) for each node
  - while traversing the tree to compute *mmv*, also compute the "best" alternative and store it in each non-leaf node
  - when looking for the best path, compare by *maxutil* when ties for *mmv*
- **Q2**: does it matter which choice?
  - No! assuming *MIN* is smart, he will always choose the smallest payoff out of all
  - unless *MIN* is not so smart

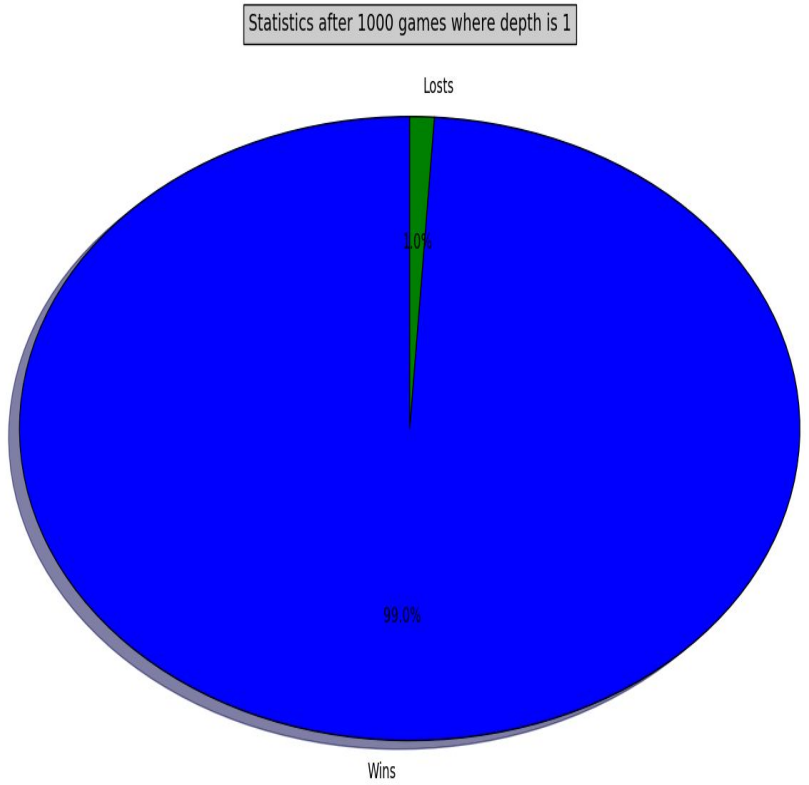


Task 2.3:Minmax search for *connect four*

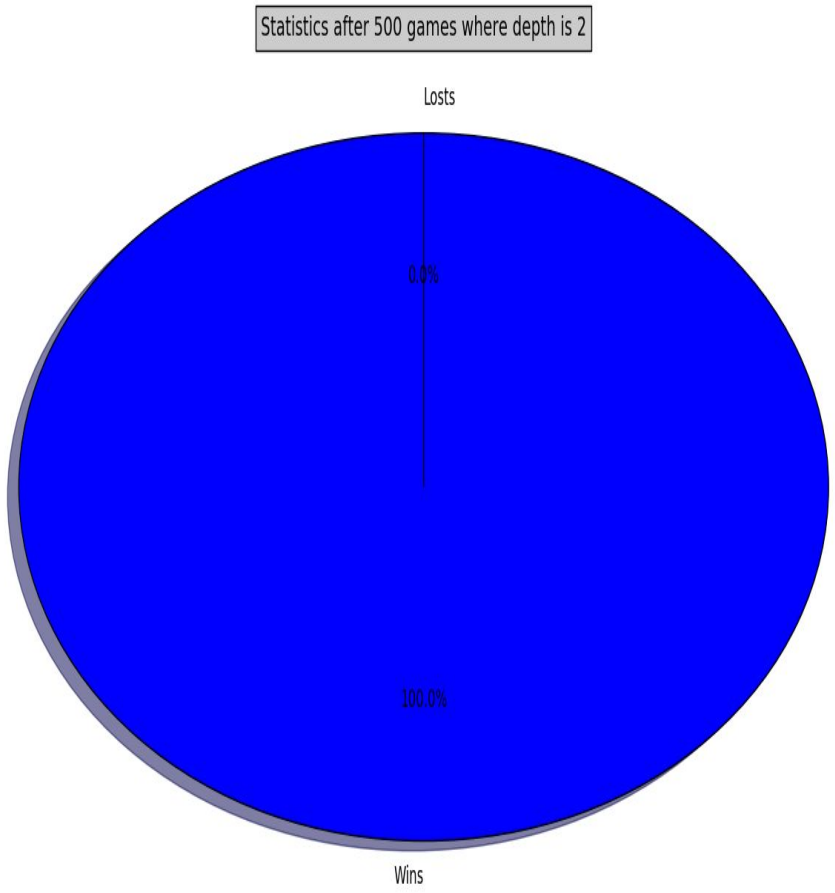
1) **Implement the depth-restricted search to find the best move**
   a) Modify depth-parameter to find the best case
2) **Implement an appropriate evaluation function**
   a) check every row,column and diagonals for possibility to put four in together and gather achieved result
   b) check every row,column and diagonals for possibility of the opponent to put four in together and gather achieved result
   c) return yourValue - opponentValue and compare with all other values to make a move

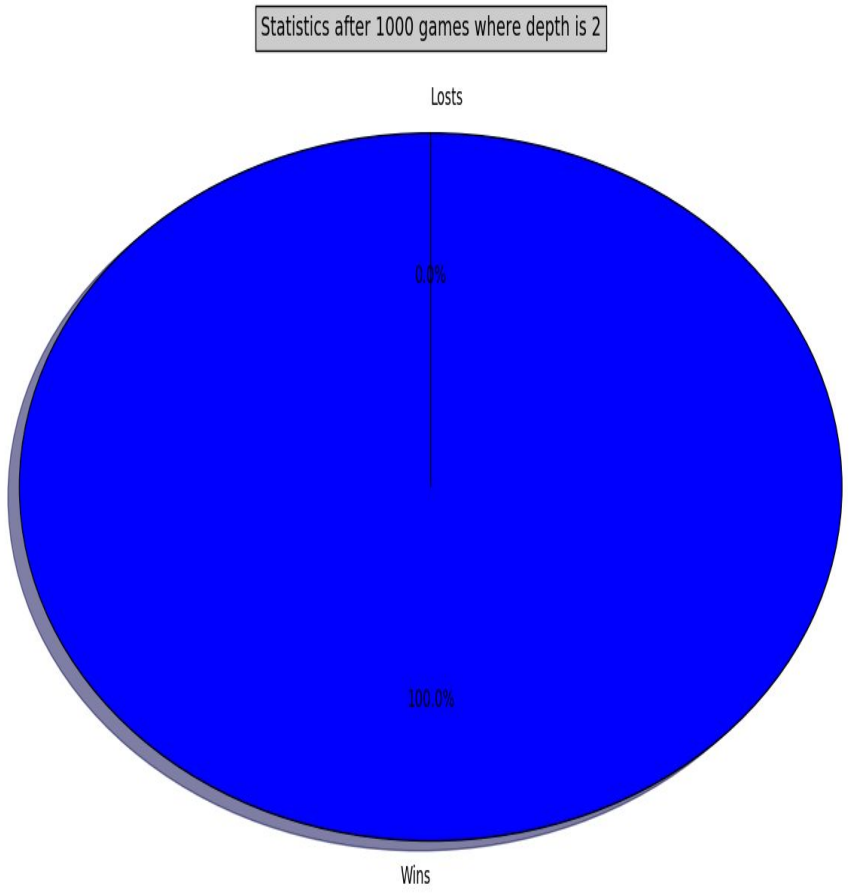
Task 2.3:Statistics (depth 1)



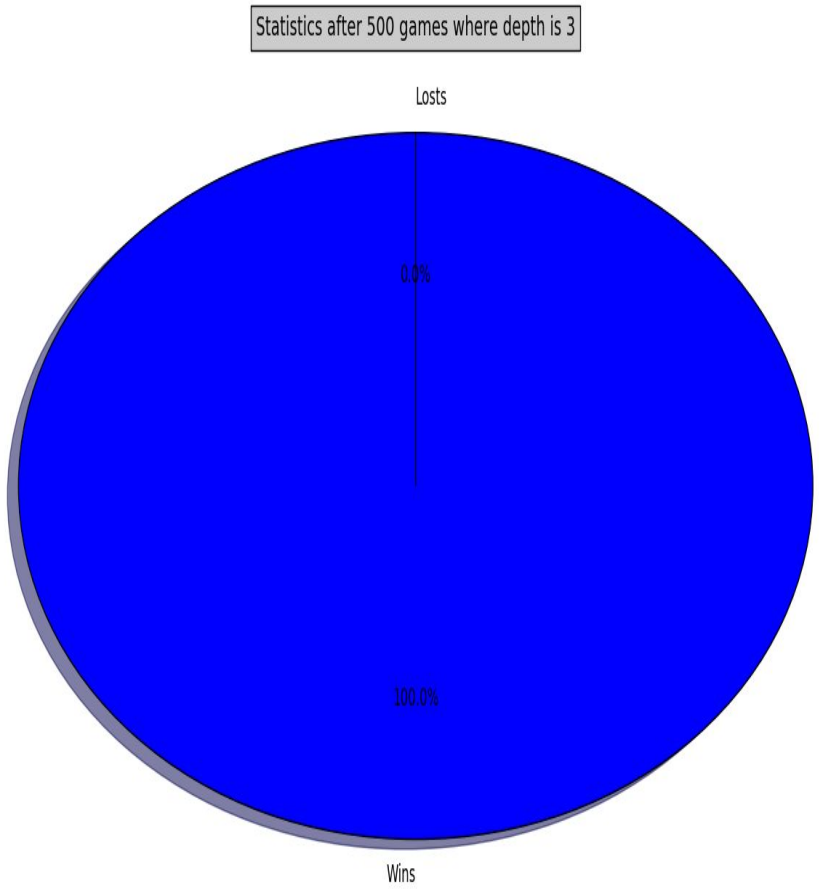


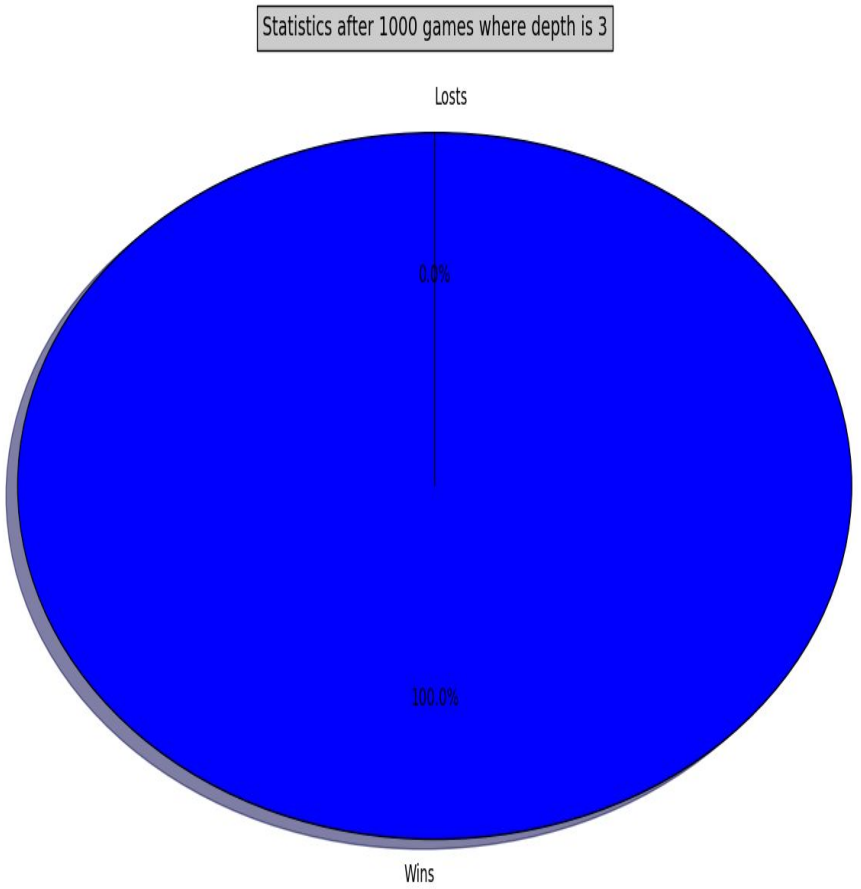
Task 2.3:Statistics (depth 2)



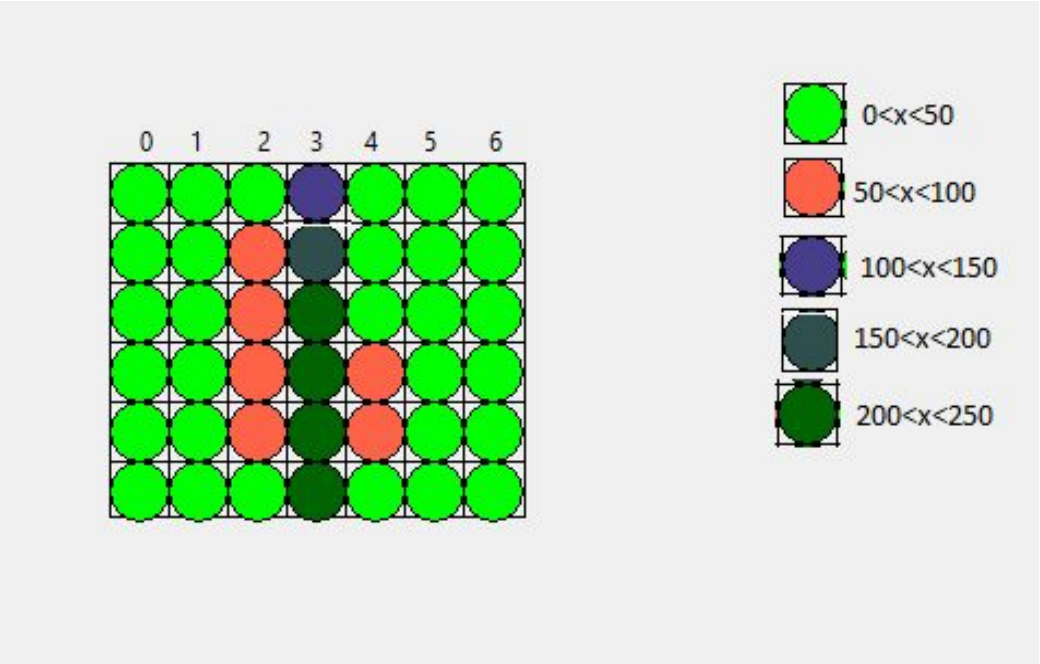


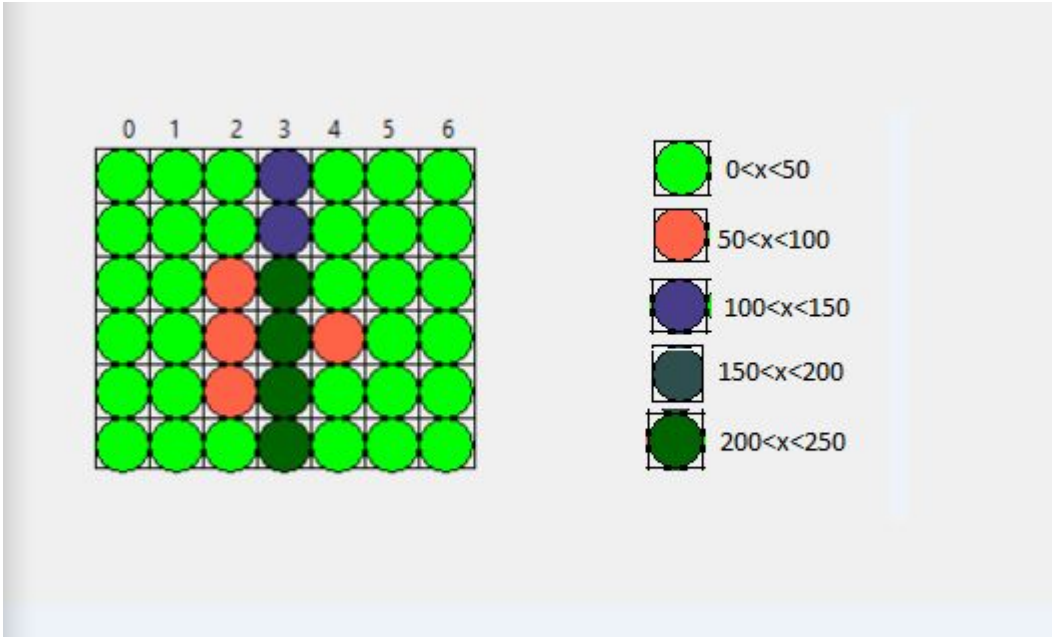
Task 2.3:Statistics (depth 3)



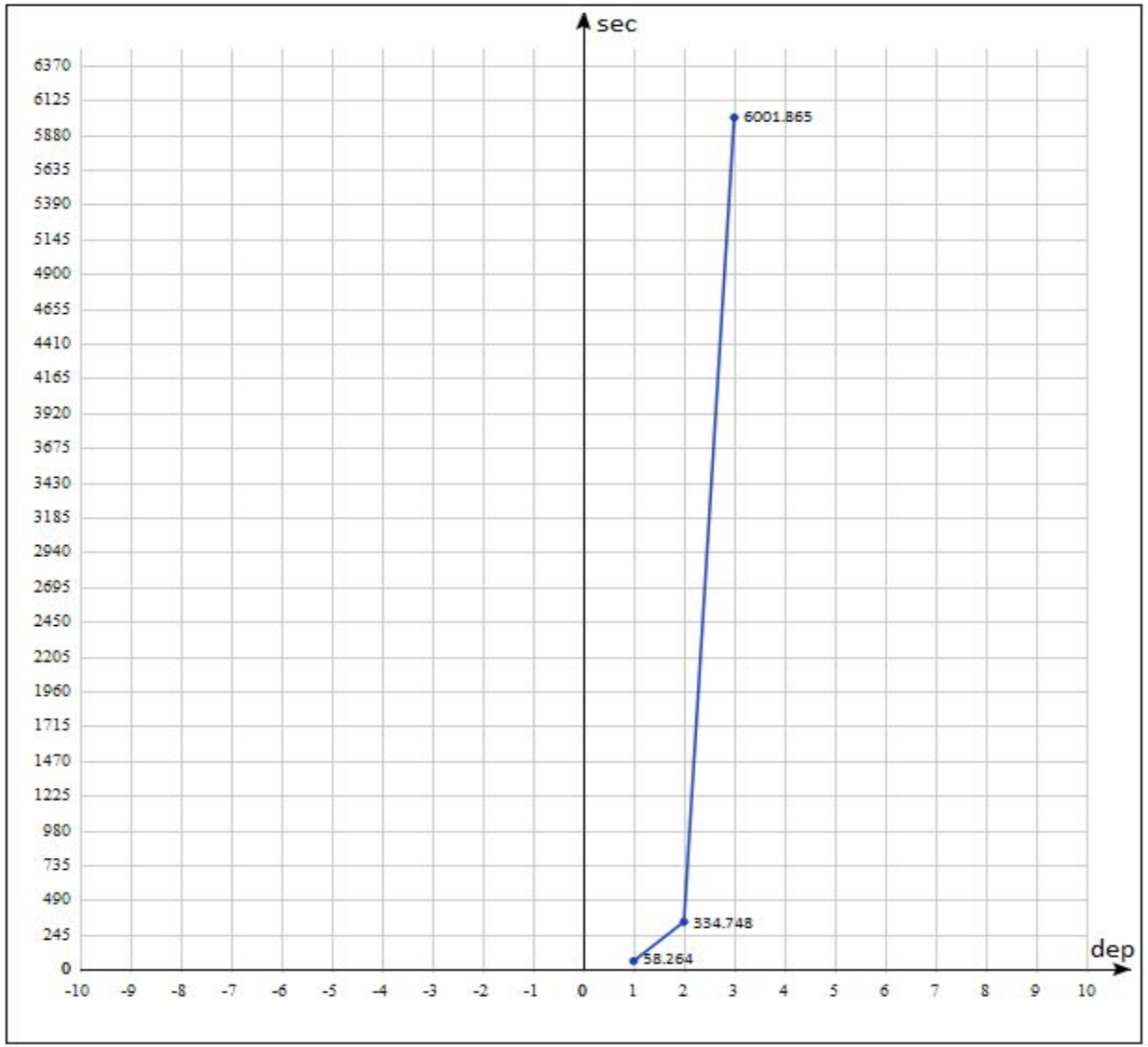


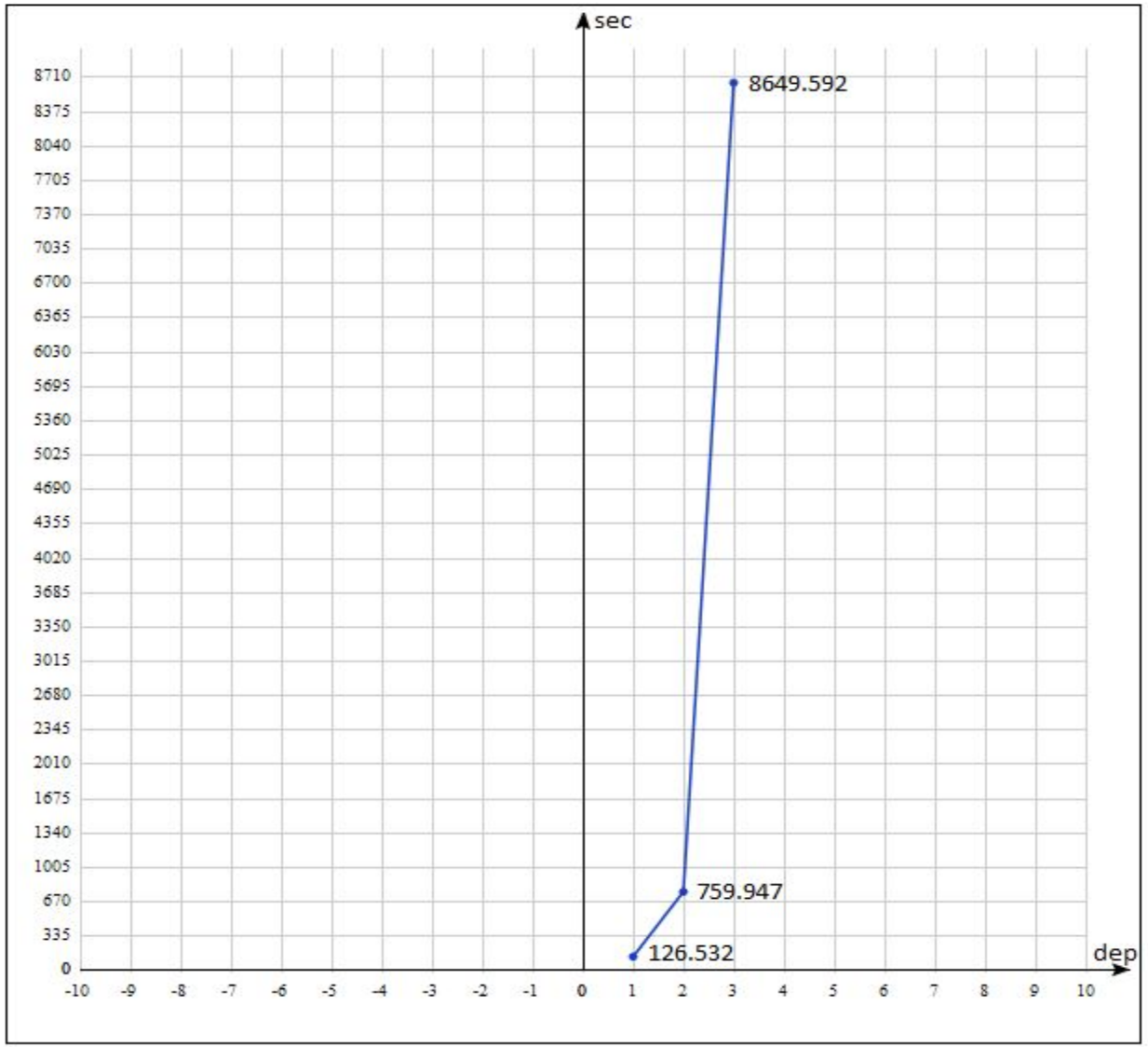
Task 2.3:Statistics (Used layers)



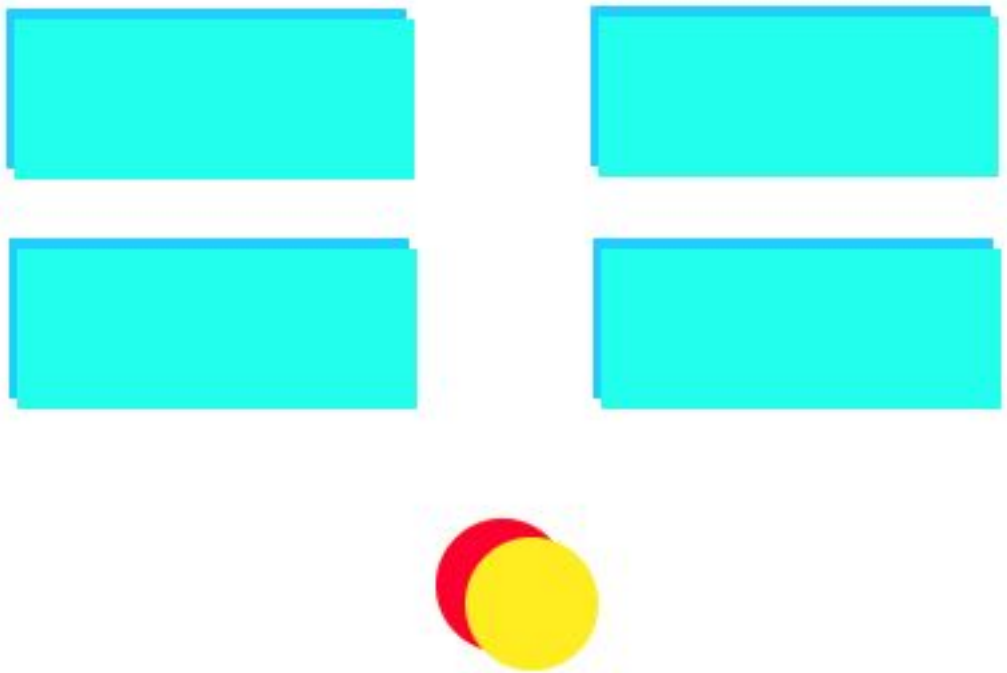


Task 2.3:Statistics (Time statistics)





Task 2.4: Breakout Predictor



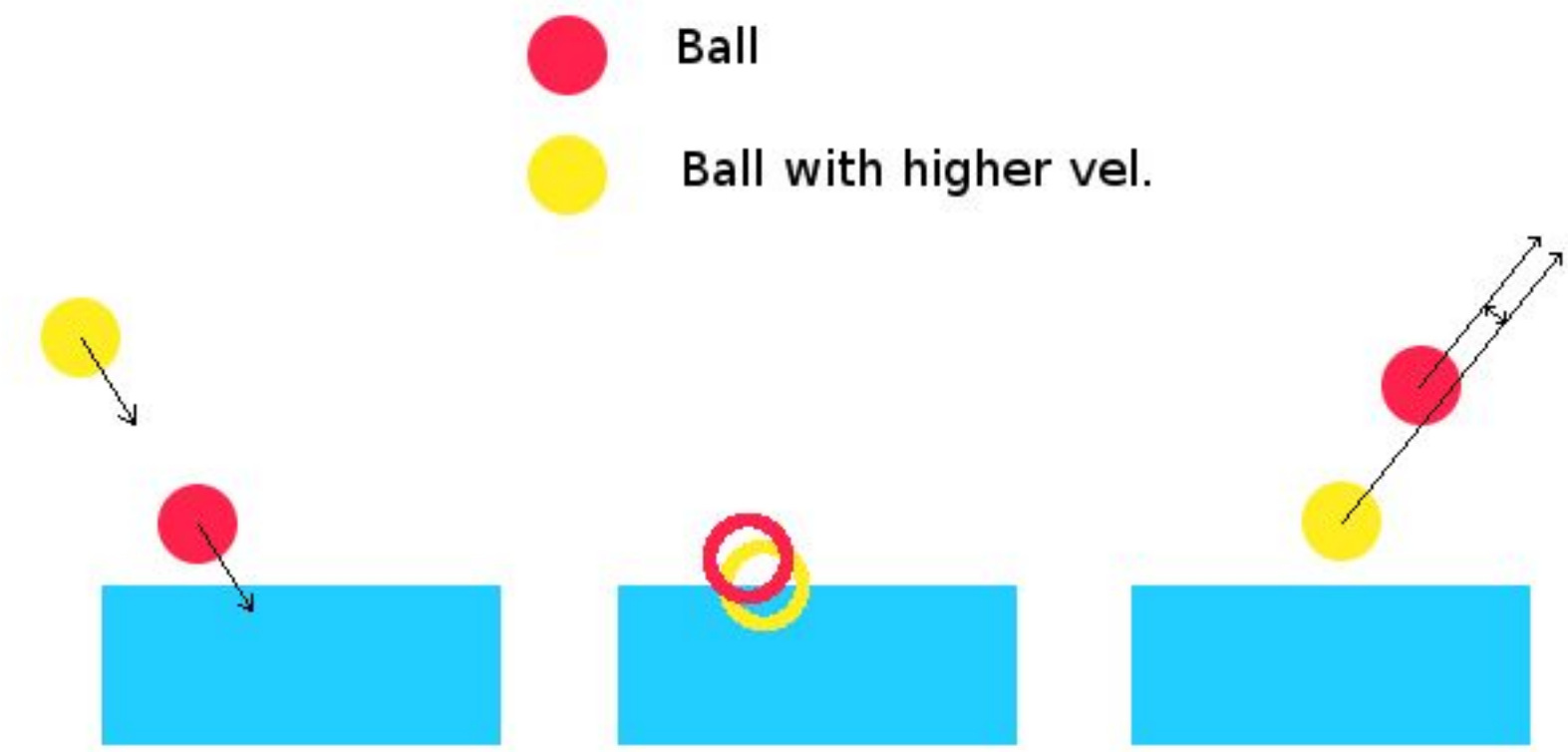
Create Copy of Ball + Bricks

Copy of ball: “predictor”

Task 2.4: Breakout Predictor

- The predictor starts with same vel_x and vel_y as ball
- Predictor moves faster than ball
- = move_predictor and collision_predictor function called 300 times per frame
- Predictor velocity not increased by increasing velocity
- Paddle follows predictor

Task 2.4: Breakout Predictor



Task 2.4: Breakout Predictor

- Ball Velocity increase over time:
- Predictor initialized every frame with same ball_vel each frame
- Paddle follows ball similarly
- Advantage: Works even for collisions causing random velocty changes
- Overkill

Task 2.5: Path Planning

Task: Given the input map, build a route from start to stop using 2 algorithms (Dijkstra and A*)

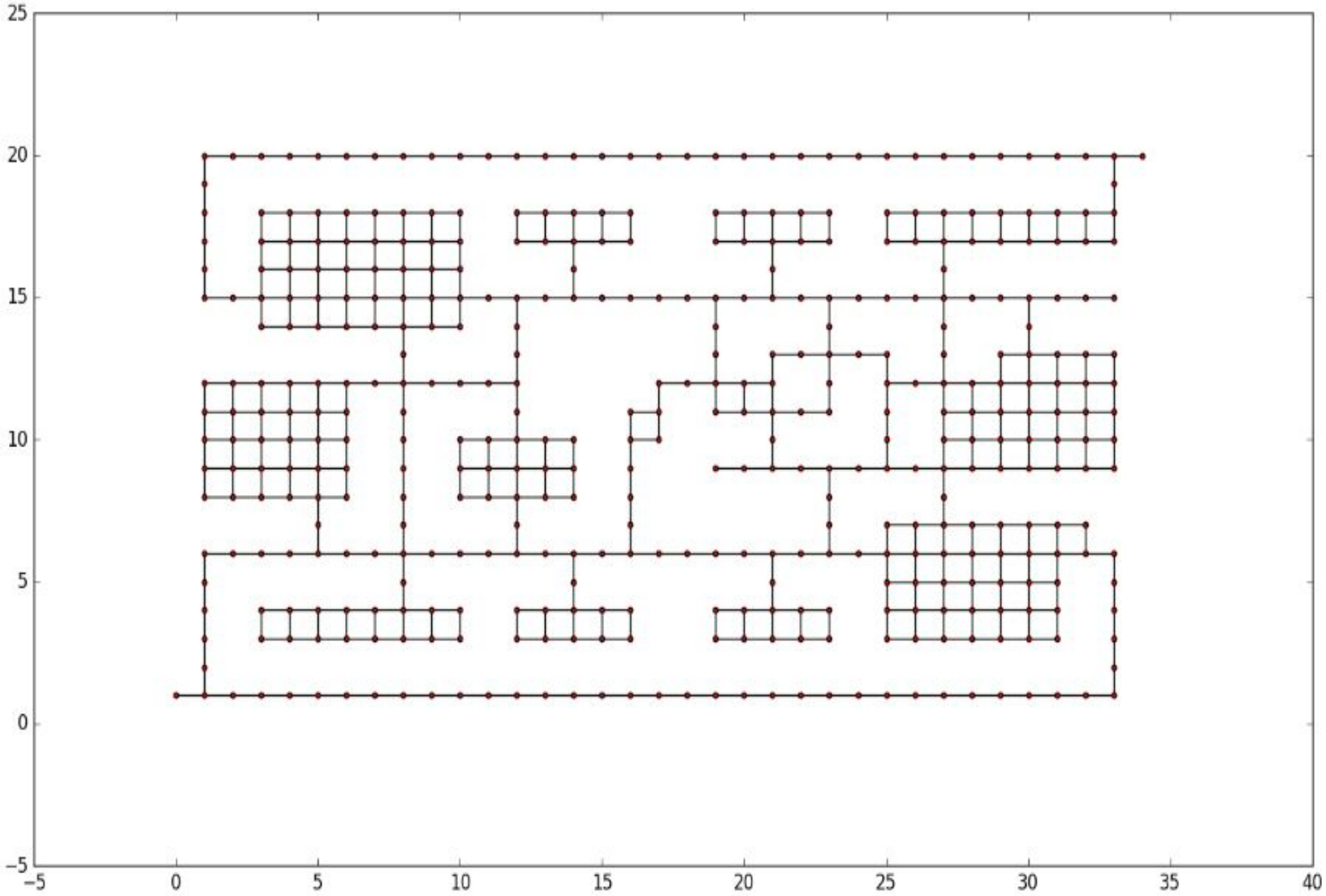
Solution:

1) Use networkx graph as data type and also visualization capabilities
2) Straight forward algo implementation

Test values: start = (1,1), stop = (33,20)

Task 2.5: Path Planning

The input



Task 2.5: Path Planning

Dijkstra
RECAP (from the lecture):

function $Dijkstra(G, s)$

$$d[v] \leftarrow \begin{cases} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{cases}$$

$p[v] \leftarrow -1$
$closed \leftarrow \emptyset$
$fringe \leftarrow V$

while $fringe \neq \emptyset$
$u \leftarrow \text{argmin} \{d[n] \mid n \in fringe\}$

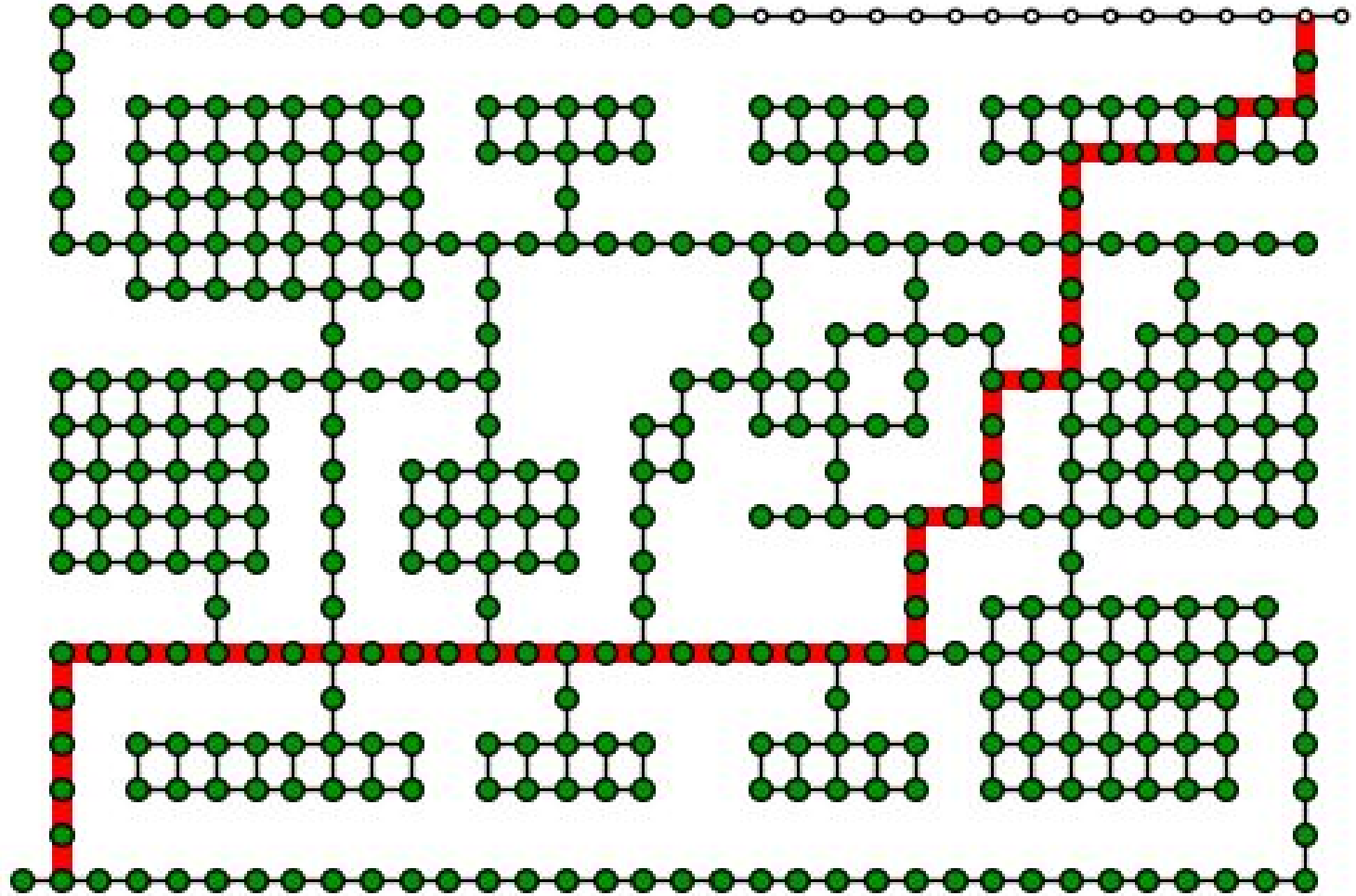
$closed \leftarrow closed \cup \{u\}$
$fringe \leftarrow fringe \setminus \{u\}$

for $v \in Neib(u) \setminus closed$
if $d[v] > d[u] + w_{uv}$
$d[v] \leftarrow d[u] + w_{uv}$
$p[v] \leftarrow u$

return d, p

Task 2.5: Path Planning

Dijkstra output



Task 2.5: Path Planning

A* algorithm
RECAP (from the lecture):

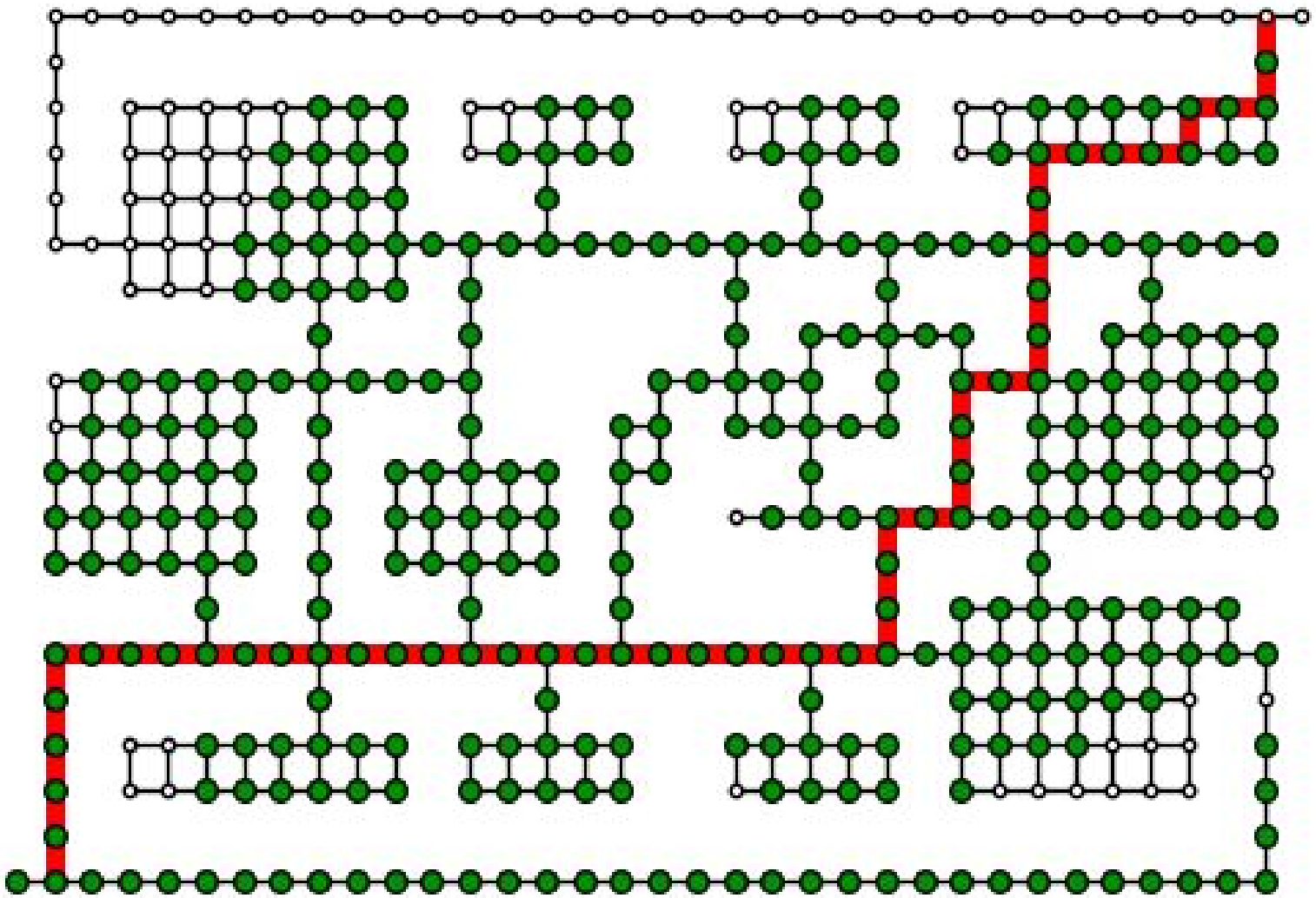
function $A^*(G, s, t)$
  $closed \leftarrow \emptyset$
  $fringe \leftarrow \{s\}$
  $g[s] \leftarrow 0$
  $f[s] \leftarrow g[s] + h(s,t)$
  while $fringe \neq \emptyset$
    $u \leftarrow \text{argmin}\{f[n] \mid n \in fringe\}$
    if $u = t$ **break**

    $closed \leftarrow closed \cup \{u\}$
    $fringe \leftarrow fringe \setminus \{u\}$

    for $v \in Neib(u) \setminus closed$
      $newg \leftarrow g[u] + w_{uv}$
      if $v \notin fringe \vee g[v] > newg$
        $g[v] \leftarrow newg$
        $f[v] \leftarrow g[v] + h(v,t)$
        $p[v] \leftarrow u$
        if $v \notin fringe$
          $fringe \leftarrow fringe \cup \{v\}$
  return g, p

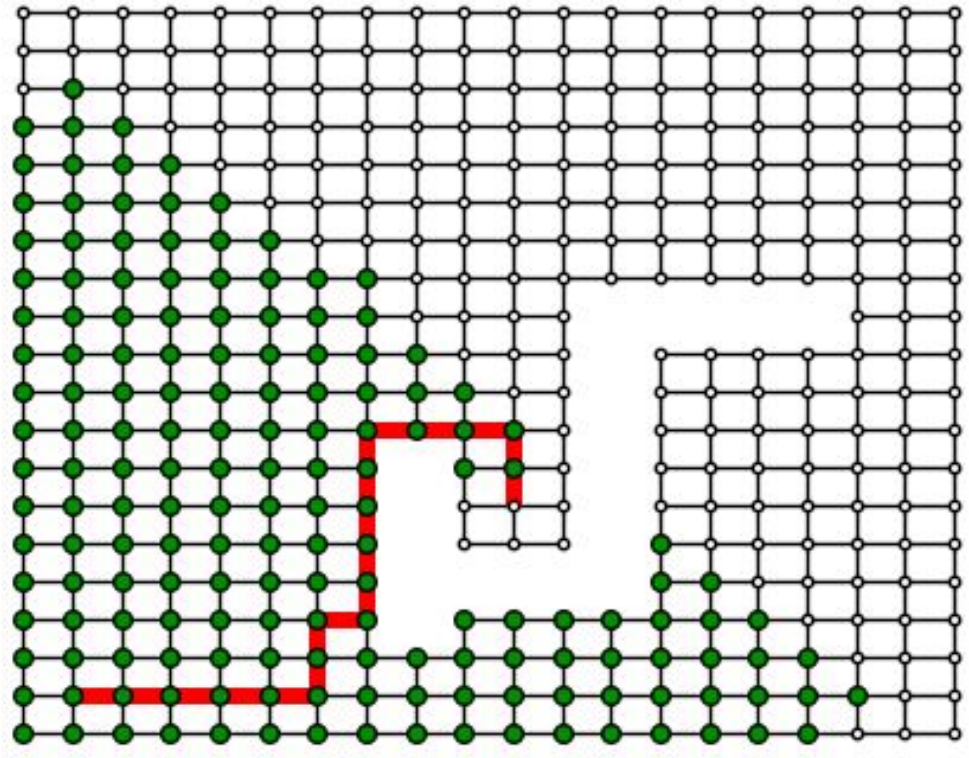
Task 2.5: Path Planning

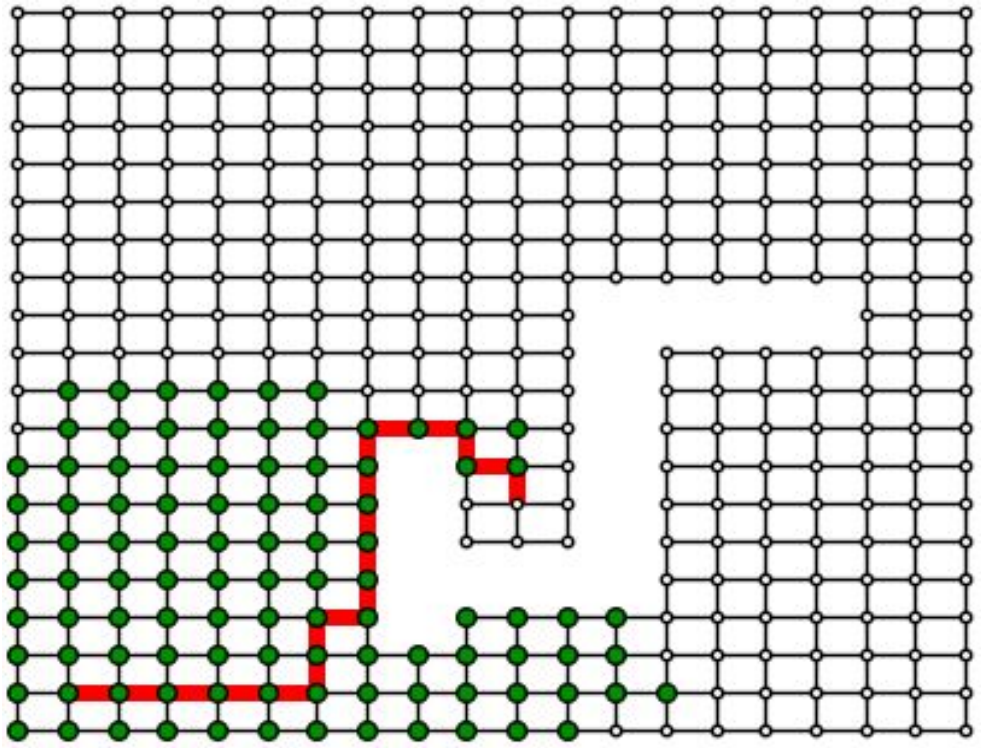
A* output



Task 2.5: Path Planning

- Extra comparison
- Dijkstra A*





Task 2.5: Path Planning

- Extra comparison
- Dijkstra A*