Implementação

- **student.py:**
  - Código de comunicação com o servidor.
  - Quando uma peça nova chega, obtém as rotações possíveis para essa peça enviando comandos ‘w’ para o servidor. Estas são guardadas num dicionário presente como atributo na classe Bot (known_rotations), onde as chaves são objetos Piece (do módulo bot.py) e os valores são tuplos com a rotação (um objeto Piece) e a sua posição quando entra em campo (apenas a horizontal importa).
  - Recorre a pesquisa em árvore para encontrar a melhor jogada. Acede ao plano da solução e envia as teclas, começando pelas rotações e depois pelas translações, incluindo um ‘s’ no fim se não comprometer o tempo de pesquisa (no início do jogo).
  - Aproveita as ações calculadas para as peças seguintes (lookahead) na pesquisa em árvore (comprometer melhor solução por desempenho).
  - Tenta garantir que todas as teclas enviadas são registadas pelo servidor.
- **bot.py:**
  - Classes para facilitar cálculos e armazenamento de dados:
    - **Bot**: armazena as rotações, dimensões do jogo, e contém outras funções úteis
    - **TetrisState**: representa o estado do jogo. Armazena o jogo, a peça corrente e as seguintes (semelhante ao ‘state’ enviado pelo servidor, excluindo o ‘game_speed’)
    - **TetrisObject**: representa um objeto genérico interativo do jogo. É a superclasse de **Piece** e **Game**, que representam as suas respetivas entidades (cada uma com as suas operações)
  - Representação binária em vez de lista de posições (um número representa uma linha, com bits a 1 onde está preenchido)
  - Reciclagem de referências de TetrisObject, que permitem evitar fazer operações repetidas (banks)

Pesquisa em Árvore

- Código reaproveitado das aulas práticas.
- Calcula as posições possíveis para cada peça, com lookahead máximo e pruning (mantém só os X melhores nodes em *open_nodes*). A pesquisa é A*. O número de moves pesquisadas e o lookahead variam com a altura e velocidade do jogo. O agente não demora mais que Y frames a efetuar uma pesquisa, terminando-a prematuramente caso isso aconteça.
- Os atributos do domínio são:
  - **Solução**: se a altura máxima do jogo neste momento é menor que a altura máxima do jogo da raíz da árvore
  - **Custo**: maior quanto menos linhas forem limpas, e vice-versa
  - **Heurística**: (explicada no slide a seguir)
  - **Ação**: número de rotações e número de translações, onde o sinal indica o sentido (negativo=esquerda, positivo=direita)
- Quando uma solução não é encontrada, envia como solução o nó com menor custo A* até o momento (BEST_EFFORT)
- Recurso a diversas caches que evitam operações repetidas, aproveitando cálculos já feitos anteriormente:
  - **Limpezas**: guarda os resultados da limpeza de linhas do jogo
  - **Heurística**: guarda cálculos de heurística para um determinado jogo
  - **Ações**: guarda a lista de ações possíveis para cada peça

Heurística

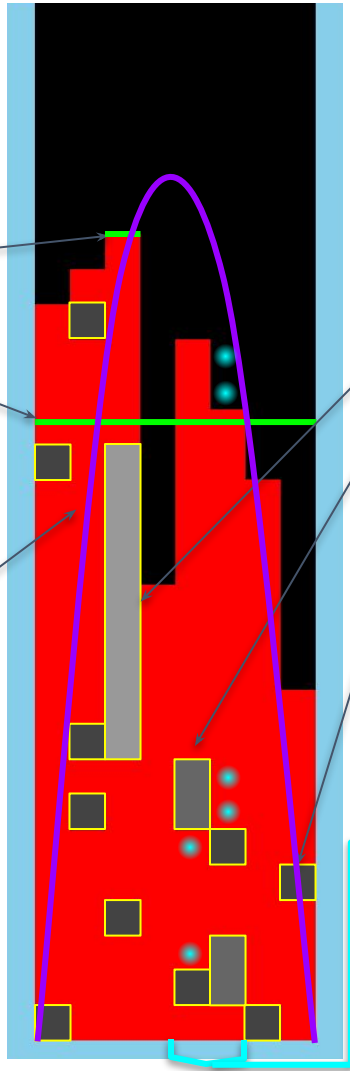
→ Altura máxima (MAX_HEIGHT)

→ Altura média (AVG_HEIGHT)

→ Variância das alturas (HEIGHT_VARIANCE): ~21

→ Ajuste da função aplicada às alturas do jogo (CENTER_SCALE), dando menor ênfase ou não às colunas laterais

$$(2 - SCALE) \cdot -\frac{\left(x - \frac{max}{2}\right)^2}{\left(\frac{max}{2}\right)^2} + 1$$



→ Buracos, analisados na vertical (HOLES)

→ Exponenciação de HOLES, dando menor ou maior importância ao tamanho dos buracos na vertical (HOLES_SCALE)

→ "Linhas limpas", avalia a diferença de altura com o jogo inicial (CLEARED_LINES)

→ Exponenciação de CLEARED_LINES, dando menor ou maior importância à magnitude da diferença de linhas (CLEARED_LINES_SCALE)

→ Continuidade, analisada na horizontal, avalia os pontos de descontinuidade nas alturas (CONTINUITY)

Melhorias

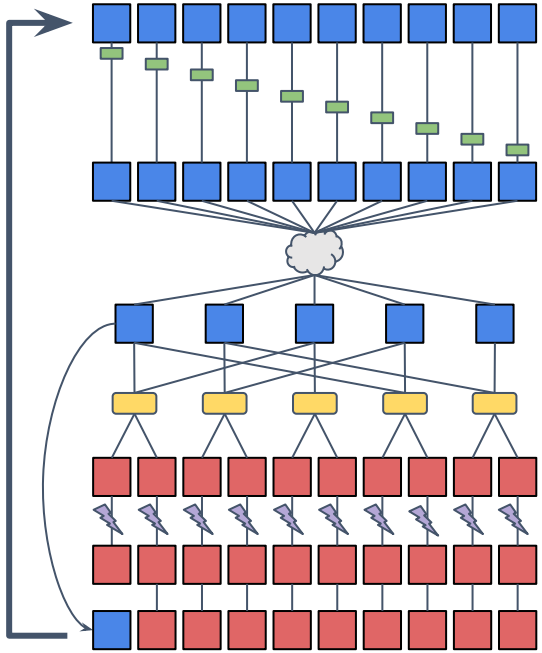
- Implementação de Tree Search
- Implementação de lookahead de 3
- Implementação de dinamicidade
- Algoritmo genético
- Criação de caches e “banks”
- Melhoria na heurística utilizada

Before

After

Algoritmo genético (melhores valores para heurística)

scripts/gen.sh



Começa com uma população de 10 indivíduos com cromossomas aleatórios (9 params * 15 bits = 135 bits), sendo que um pode ter parâmetros definidos como argumento

Fitness: média de pontuação de 10 jogos de Tetris (jogados ao mesmo tempo)

Seleção: seleção da melhor metade

Cruzamento: selecionar 5 pares, obter um par de filhos para cada com junção das partes dos cromossomas dos pais (ponto de partição aleatório)

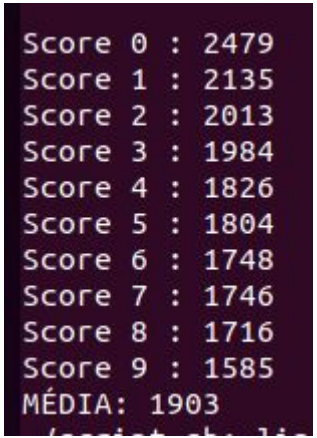
Mutação: altera um bit do cromossoma para garantir alguma divergência (100% de chance, visto que o algoritmo converge facilmente e a mutação é fraca)

Elitismo: mantém sempre o melhor indivíduo (permite que os melhores parâmetros não se percam e maior taxa de mutação)

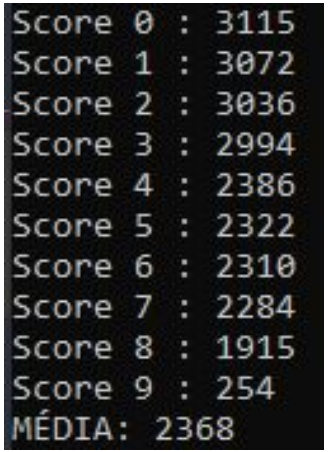
Os valores por defeito no TetrisDomain são resultado de 11 gerações, começando com média de ~1200 e acabando em média de ~1700

Resultados

Pedro
(agente corrido em Ubuntu 20.04)



Filipe
(agente corrido em Linux Manjaro)



Martinho
(agente corrido em Ubuntu 20.04)