Sokoban

Desenvolvimento de um agente autónomo

Inteligência artificial

2020 - 2021

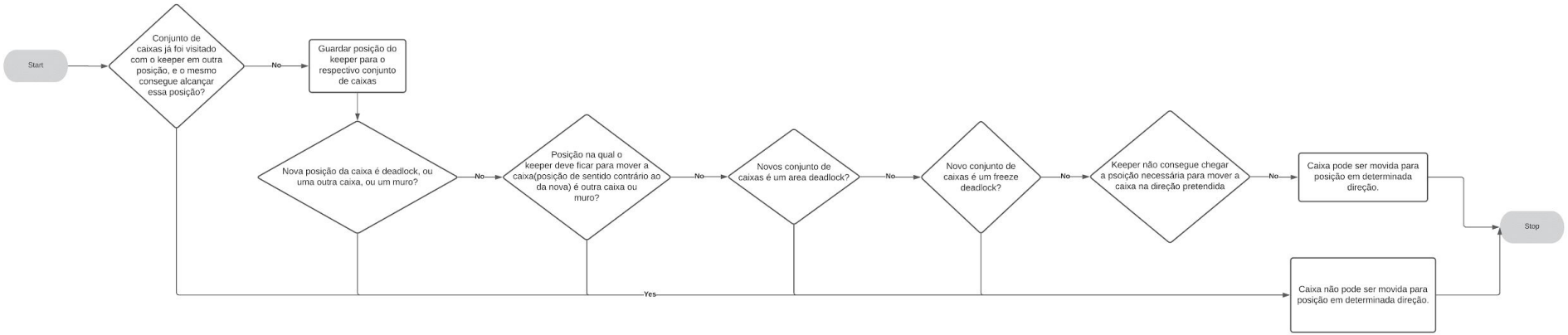
Renan Ferreira, 93168

Introdução

O presente trabalho tem como objetivo pôr em prática os conhecimentos explicados na cadeira de Inteligência Artificial do ano letivo 2020 - 2021. O trabalho envolve criar um agente autónomo capaz de resolver os variados níveis do jogo sokoban.

Diagrama de decisões do Sokoban

Determinada caixa em determinado estado(conjunto de caixas e um keeper) deve se movimentar para determinada direção?



Estratégia de pesquisa e Heurística

Estratégia de pesquisa - Depois de variados testes com diferentes valores de estratégia e heurística, conclui que a melhor a ser usada é a ***greedy***. A mesma é extremamente eficiente em termos de complexidade temporal, porém não em termos de optimalidade, demandando muitos movimentos para chegar a solução de determinado nível.

Heurística - Considerando o principal domínio envolvido na pesquisa, o de movimento de uma caixa só, definido na classe ***BoxDomain***, cheguei a usar diferentes tipos de heurística, como distância manhattan ou euclidiana, associada com algoritmo hungáro de atribuição, porém decidi me no final por usar a heurística greedy associada a distância ***Pull distance***. No início, cheguei a pensar em usar combinações de heurísticas, definindo sempre a de maior valor, porém essencialmente a última geralmente acabava tendo o maior valor e para aliviar o desempenho computacional da pesquisa,decidir por me usar só dela. Também pensei em fazer somatório da mesma com a menor distância do keeper a todas as caixas, porém o valor era insignificante na maioria das vezes para justificar o cálculo. A seguir, irei explicar com detalhes a heurística e o parâmetro de métrica de distância aplicada.

Distância *Goal Pull* - Tal métrica envolve uma pré-computação no domínio do problema. A mesma define o número de pushes necessários para mover uma caixa em determinada posição para cada goal, sem considerar a presença de outras caixas e que o keeper consegue alcançar qualquer posição no tabuleiro. No código, o mesmo o mesmo é calculado usando um algoritmo ***breadth-first*** para fazer ***push*** de cada ***goal*** para diferentes posições. Após isso, os cálculos são guardados em uma váriavel ***distanceToGoal***, que determinada a distância de um goal para determinada posição. Lembrando que já que algumas posições não conseguem alcançar determinado ***goal***, a distância de uma a outra é calculada como infinita. Abaixo, pode-se ver uma imagem da implementação do código de geração da métrica. A referência ao algoritmo é o trabalho de tese de Nils Froleyks.

Greedy Algoritmo - A escolha deste algoritmo foi baseada na pesquisa de Nils Froleyks também. Para definir a atribuição entre caixas e goals, usei o algoritmo greedy. Não irei explicar em detalhes sobre como funciona o algoritmo, pois o mesmo é melhor explicado no trabalho do acadêmico, porém pode-se dizer que escolhe os pares goal e caixa que tenham menor distância, e assinalando no final os elementos que não conseguiram ser pareados na primeira atribuição. A vantagem desse algoritmo é a possibilidade de retornar valores infinitos(ou essencialmente muito altos) para definir estados na qual o custo para o alcançar é extremamente alto ou impossível, em um método que pode ser considerado de pruning. Abaixo, é possível ver a implementação do algoritmo.

```
self.distanceToGoal = dict()
for goal in self.goals:
    tmp = dict()
    self.distanceToGoal[goal] = tmp
    for pos in (self.floor + self.goals):
        tmp[pos] = float('inf')

queue = []
for goal in self.goals:
    self.distanceToGoal[goal][goal] = 0
    queue[:0] = (goal),
    while queue != []:
        pos = queue.pop()
        for dir in directions():
            boxpos = new_pos(pos, dir)
            playerpos = new_pos(boxpos, dir)
            if(not mapa.is_blocked(boxpos) and not mapa.is_blocked(playerpos)):
                goaldict = self.distanceToGoal[goal]
                if goaldict[boxpos] == float('inf'):
                    goaldict[boxpos] = goaldict[pos] + 1
                    queue[:0] = (boxpos),
```

Pré- computação da métrica ***goal pull.***

```
def greedy_distance(self, boxes, infinite=100000000):
    edges = sorted([((goal, box), self.distanceToGoal[goal][box])
        for box in boxes for goal in self.goals], key=lambda p: p[1])
    for idx in range(len(edges)):
        edge = edges[idx]
        if edge[1] == float('inf'):
            edges[idx] = (edge[0], infinite)

    matches = []
    matchedBoxes = set()
    matchedGoals = set()
    for tmp in edges:
        goal = tmp[0][0]
        box = tmp[0][1]
        if(not (goal in matchedGoals) and not(box in matchedBoxes)):
            matches += [tmp]
            matchedBoxes.add(box)
            matchedGoals.add(goal)

    for box in boxes:
        if box not in matchedBoxes:
            closestgoal = None
            for goal in [goal for goal in self.goals if goal not in matchedGoals]:
                if (box in self.distanceToGoal[goal]) and (
                    closestgoal is None or self.distanceToGoal[goal][box] < self.distanceToGoal[closestgoal][box]):
                    closestgoal = goal
            matches += [((closestgoal, box), self.distanceToGoal[closestgoal][box])]
            matchedBoxes.add(box)
            matchedGoals.add(closestgoal)

    return sum([idx[1] for idx in matches])
```

Algoritmo de atribuição ***greedy***.

Detecção de Deadlocks

Um importante fator de pruning durante a execução da pesquisa, deadlocks são estados na qual não é possível obter soluções. Diferentes deadlocks podem ser detectados com diferentes técnicas. Neste projeto, os deadlocks que foram analisados foram os:

1. Static deadlocks
2. Freeze deadlocks
3. Area deadlocks

```
def deadlock_detection(self, boxes, box, direction):
    newboxes = self.get_newboxes(boxes, box, direction)
    if self.areadeadlock_detection(newboxes):
        return True

    newbox = new_pos(box, direction)
    if not newbox in self.goals:
        if self.freeze_deadlock_detection(newboxes, self.walls, newbox):
            return True
    return False
```

Static Deadlock

Sua implementação foi baseada, juntamente com os outros deadlocks, no site sokobano.de. São posições na qual se qualquer caixa for colocada lá, não conseguem alcançar nenhum goal. Se usa das métricas do distanceToGoal, na qual para qualquer posição, se sua distância for infinita para todos os goals, então a mesma é guardada em uma varíavel chamada simpledeadlocks. Assim, o domínio que nenhuma caixa pode ser colocada lá.

```
def is_movable(self, boxes, walls, box, direction):
    obstacles  = self.get_other_boxes(boxes, box) + walls
    newbox = new_pos(box, direction)
    return inside_range(newbox, self.size) and (
        newbox not in obstacles + self.simpledeadlocks) and (
        prior_pos(box, direction) not in obstacles)
```

Freeze Deadlock

Para analisar se determinada caixa está em freeze deadlock, é preciso checar em cada uma das 4 direções pelos seguintes fatores:

1. Se existe um muro
2. Se existe um static deadlock
3. Se existe uma outra caixa e se essa caixa também estiver em freeze deadlock.

Se todos se confirmarem verdadeiros, então existe um freeze deadlock. O algoritmo usa uma estrutura recursiva para analisar as caixas adjacentes e vê se as mesmas também estão em freeze deadlock.

```
def freeze_deadlock_detection(self, boxes, walls, box):
    if len([dir for dir in directions() if self.is_movable(boxes, walls, box, dir)]) == 0:
        countbox      = 0
        countdeadlock = 0
        for dir in directions():
            newbox = new_pos(box, dir)
            if(newbox in boxes):
                countbox += 1
                newboxes = self.get_other_boxes(boxes, box)
                newwalls = walls + [box]
                if self.freeze_deadlock_detection(newboxes, newwalls, newbox):
                    countdeadlock += 1
        if(countbox == countdeadlock):
            return True
    return False
```

Area Deadlock

Baseado na pesquisa de Nils, a análise desse deadlock envolve uma certa pré-computação. Usando os valores em distanceToGoal, é definido um conjunto de goals com uma respectiva lista que diz quais posições são alcançaveis para a mesma. Este valor é guardado em areas. Caso o número de caixas seja maior que o número de posições para cada conjunto de goals em areas, então é detectado um deadlock.

```
reachablegoals = dict()
self.areas = dict()

for pos in self.floor:
    if pos not in self.simpledeadlocks:
        reachablegoals[pos] = []

for pos in reachablegoals:
    for goal in self.goals:
        if self.distanceToGoal[goal][pos] != float('inf'):
            (reachablegoals[pos]).append(goal)
    goals = frozenset(reachablegoals[pos])
    reachablegoals[pos] = goals
    if goals not in self.areas:
        self.areas[goals] =  [pos]
    else:
        self.areas[goals] += [pos]
```

```
def areadeadlock_detection(self, boxes):
    for area in self.areas:
        if sum([1 for box in boxes if box in self.areas[area]]) > len(self.areas[area]):
            return True
    return False
```

Outros métodos de pruning

Além da detecção de deadlocks, outros mecanismos foram usados para diminuir o espaço de pesquisa e melhorar sua eficiência. São estes:

Estados visitados - Estados já visitados geram um hash que é guardado na classe treeSearch para diminuir o espaço de busca.

Estados que são alcançaveis a partir de estados já visitados - Para melhorar a pesquisa, é possível filtrar estados que são visitaveis a partir de outros estados. No BoxDomain, existe uma váriavel, visitedkeepers, que para cada conjunto de caixas, vê se o mesmo já foi explorado anteriormente, mesmo que o keeper estivesse em outra posição. Se for comprovado que o keeper é capaz de alcançar a sua posição anterior, significa que a exploração deste novo estado não é necessária. A classe retorna o valor -1, que significa visitável, para assim o TreeSearch inserir no conjunto de hashs já visitados.

```
def visitable(self, state):
    key = frozenset(state[1])
    if key in self.visitedkeepers:
        for visitedpos in self.visitedkeepers[key]:
            plan = self.keeper_plan(state[1], state[0], visitedpos)
            if not plan is None:
                return True
    return False
```

```
def actions(self,state):

    if self.visitable(state):
        return -1

    if frozenset(state[1]) in self.visitedkeepers:
        self.visitedkeepers[frozenset(state[1])].append(state[0])
    else:
        self.visitedkeepers[frozenset(state[1])] = [state[0]]

    actlist = []
    for box in state[1]:
        for direction in [dir for dir in directions() if self.allowed(state, box, dir)]:
            plan = self.keeper_plan(state[1], state[0], prior_pos(box, direction))
            if not plan is None:
                actlist += [(box, plan + [direction])]
    return actlist
```

O que faltou?

Irei exemplificar o que gostaria de ter melhorado porém não tive tempo:

1. Estrutura de decisão sobre a estratégia de pesquisa, para que assim, dependendo da média de ramificação da pesquisa, a mesma podendo ser mudada a meio para que assim se alcançasse os requisitos adequados principalmente a optimalidade e complexidade temporal.
2. Implementar o pi-corral pruning.

Resultados

Os resultados obtidos mostram um bom desempenho em termos de complexidade temporal, porém as soluções não são as melhores possíveis para o problema. O sokoban consegue resolver até o nível 130. Como dito anteriormente, através de experimentos com diferentes estratégias e heurísticas, chegou-se a conclusão que a melhor solução é usar estratégia greedy associada com algoritmo greedy de atribuição e métrica goal pull. Os resultados são eficientes em termos de tempo, porém nem sempre são os melhores.

Referências

Using an Algorithm Portfolio to Solve Sokoban, Nils Froleyks:

https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf

How to detect deadlocks:

http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks