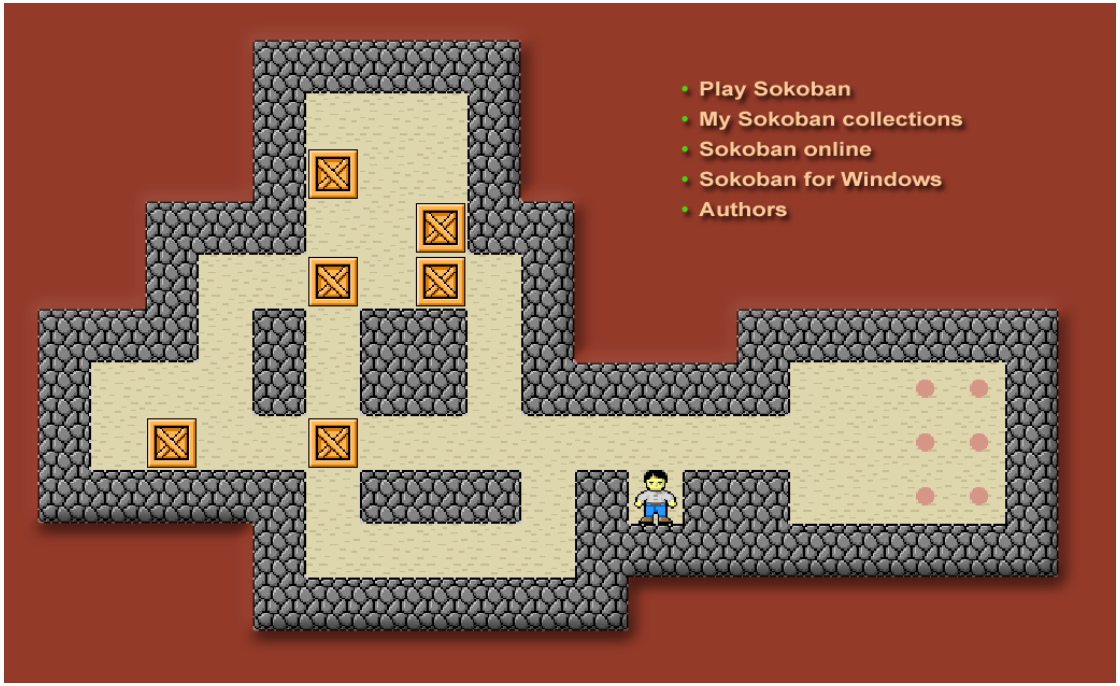
倉庫番になろう

藤原 裕大

倉庫番とは

- 1982 年にシンキングラビット社が発売したゲーム
- プレイヤーを操作して、箱を格納場所に移動する



倉庫番とは

なんかすごい難しいらしい

Sokoban can be studied using the theory of computational complexity. The problem of solving *Sokoban* puzzles has been proven to be NP-hard.[5][6] Further work showed that it was significantly more difficult than NP problems; it is PSPACE-complete.[7] This is also interesting for artificial intelligence researchers, because solving *Sokoban* can be compared to the automated planning that needs to be done by a robot that moves boxes in a warehouse.

Sokoban is difficult not only due to its branching factor (which is comparable to chess), but also its enormous search tree depth; some levels can be extended indefinitely, with each iteration requiring an exponentially growing number of moves and pushes.[8] Skilled human players rely mostly on heuristics; they are usually able to quickly discard futile or redundant lines of play, and recognize patterns and subgoals, drastically cutting down on the amount of search.

Some *Sokoban* puzzles can be solved automatically by using a single-agent search algorithm, such as IDA*, enhanced by several techniques which make use of domain-specific knowledge.[9] This is the method used by *Rolling Stone*,[10] a *Sokoban* solver developed by the University of Alberta GAMES Group. The more complex *Sokoban* levels are, however, out of reach even for the best automated solvers.[11]

深さ優先探索 (DFS)

- やるだけ

```
def solve(初期状態)
    stack ← {初期状態}
    visited ← Φ
    while stackが空でない
        s ← stackからpop
        if sがvisitedに含まれる then continue endif
        visited ← visited + {s}
        if sが目的状態 then break endif
        if sがデッドロック then continue endif
        stack ← stack + 考えられる次の状態集合
    endwhile
    if sが目的状態 then return 移動経路
    else 探索失敗 endif
enddef
```

反復深化深さ優先探索 (IDDFS)

- やるだけ

```
def solve(初期状態)
    bound ← h(初期状態)
    stack ← {初期状態}
    loop
        visited ← Φ
        t ← search(stack, 0, bound, visited)
        if t が解発見 then break endif
        if t が探索失敗 then 探索失敗 endif
        bount ← t
    endloop
enddef

def search(stack, cost, bound, visited)
    node ← stack から pop
    f ← cost + h(node)
    if node が visited に含まれる then return 探索失敗 endif
    visited ← visited + {node}
    if f > bound then return f endif
    if node が目的状態 then return 解発見 endif
    minimum ← 無限大
    for s in 考えられる次の状態
        stack ← stack + {s}
        t ← search(stack, cost+1, bound, visited)
        if t が解発見 then return t endif
        if t < minimum then minimum ← t endif
        stack から pop
    endfor
    return minimum
enddef
```

A* 探索 - 評価関数の問題

ヒューリスティック関数をどう設定する？

```
def solve(初期状態)
    queue ← {初期状態} (優先度付きqueue)
    while queueが空でない
        s ← queueから取り出す
        if sがvisitedに含まれる then continue endif
        visited ← visited + {s}
        if sが目的状態 then break endif
        if sがデッドロック then continue endif
        queue ← queue + 考えられる次の状態集合
    endwhile
enddef
```

A* 探索 -h(x) を定義する

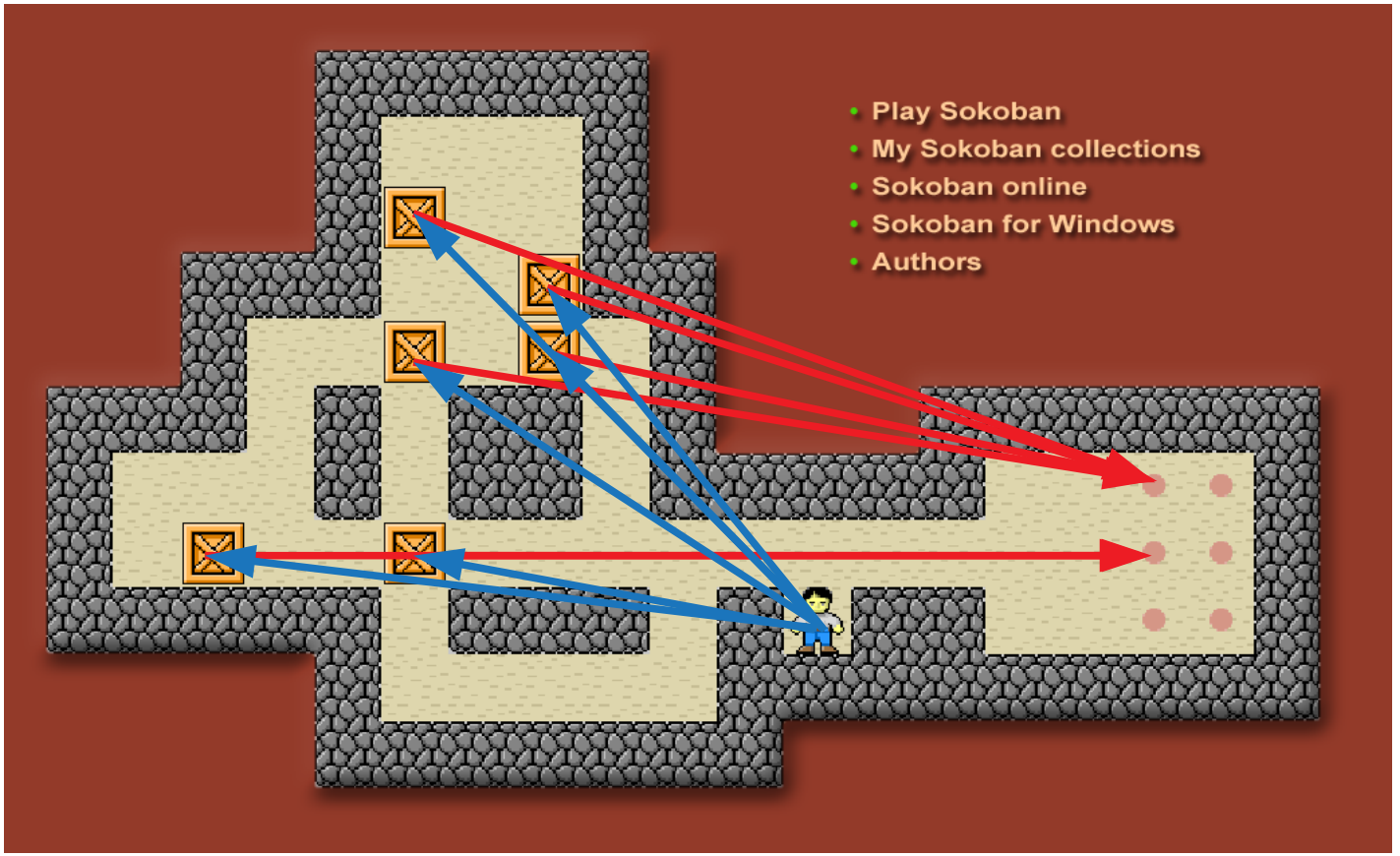
今回は推定関数として箱とゴールの距離を使った

$$h(x) = \sum_i d(b_i, g_{argmin_j(d(b_i,g_j))})$$

「各箱について最も近いゴールとの距離」の総和

h(x)

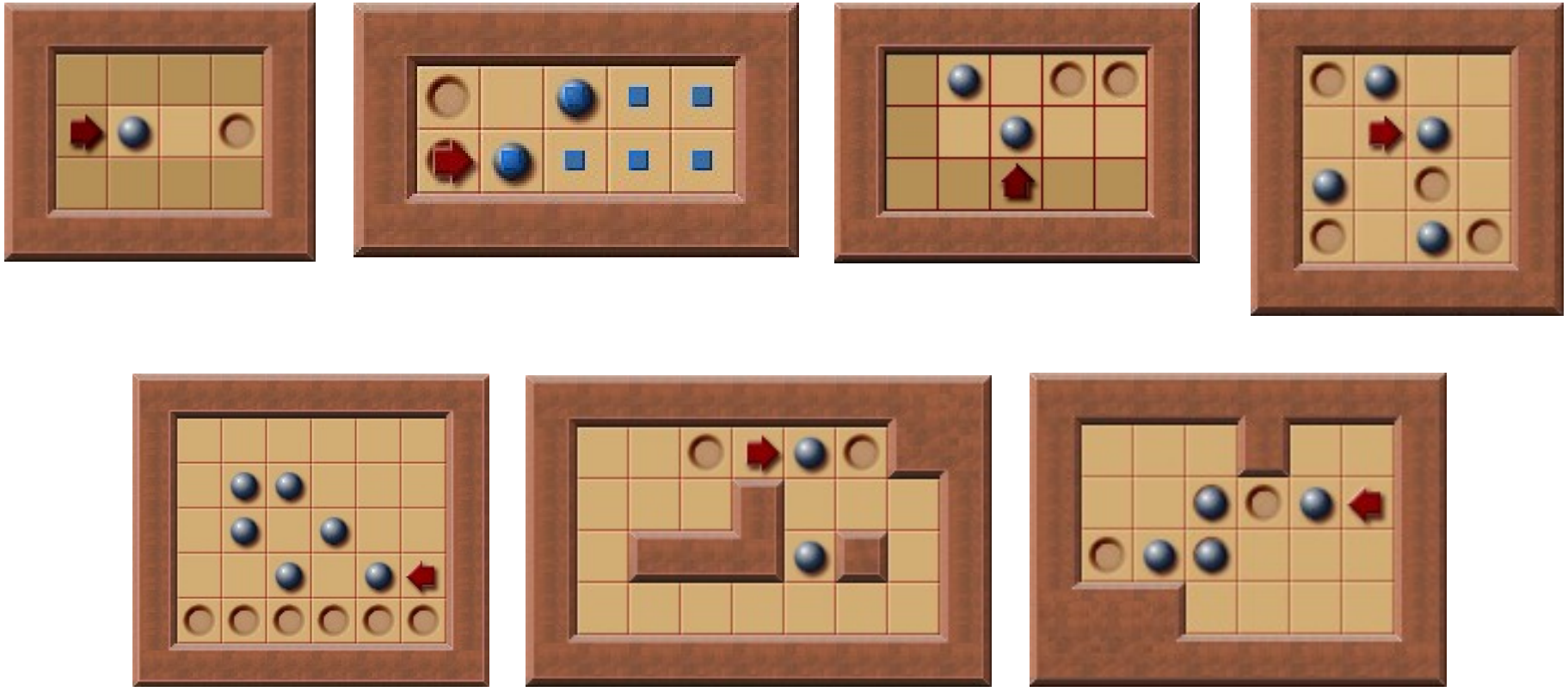
実際のコストより明らかにたぶん許容的だと思う



デッドロック

- 箱をある位置に動かしてしまうと解けなくなる
- デッドロックの種類
  - Dead square deadlocks
  - Freeze deadlocks
  - Corral deadlocks
  - Closed diagonal deadlocks
  - Bipartite deadlocks
  - Deadlocks due to frozen boxes

デッドロック



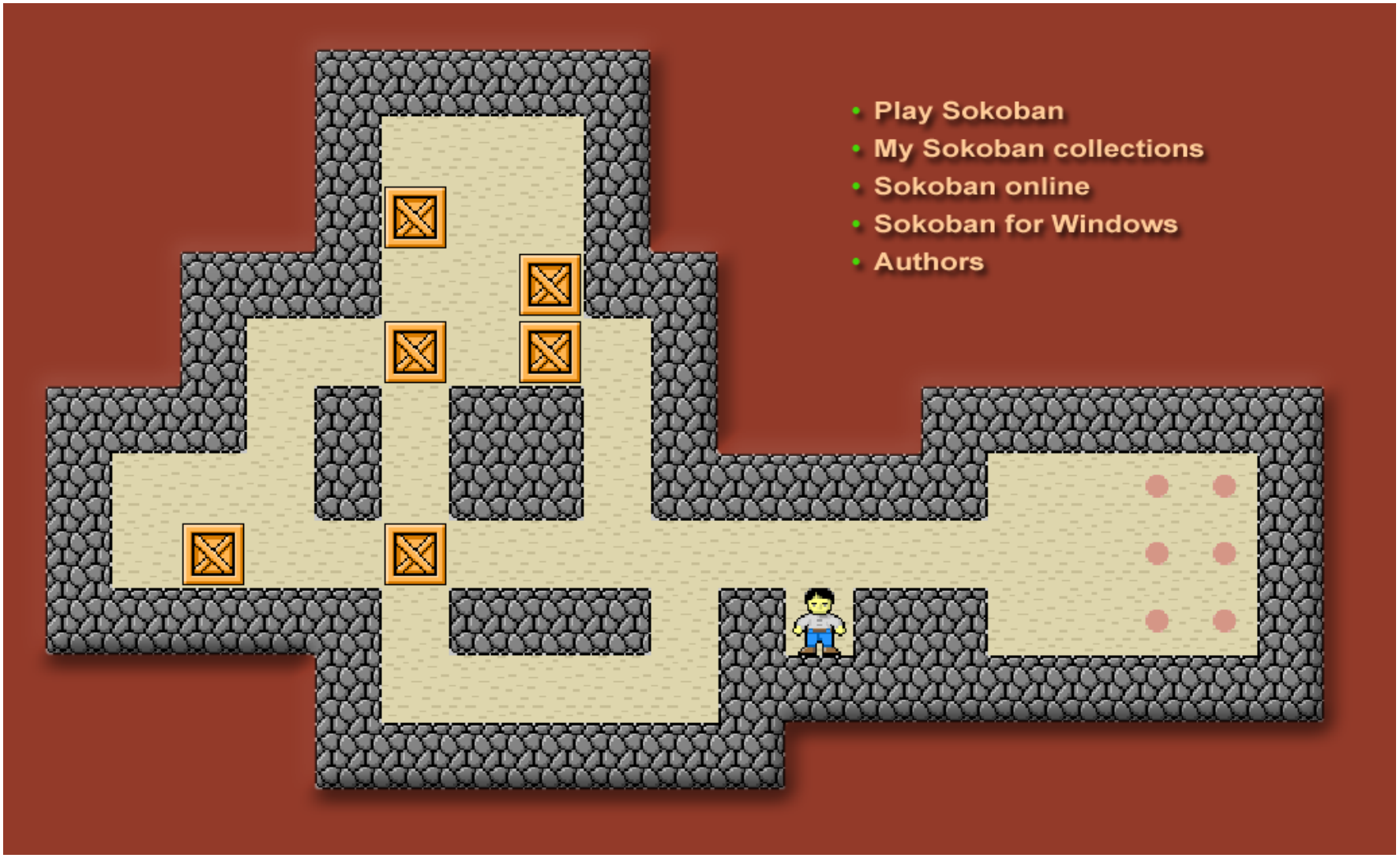
デッドロックを判定したい

- 静的解析と動的解析により潰す
  - Dead square deadlocks → 静的解析
  - Freeze deadlocks → 動的解析
  - Corral deadlocks → 検出しない
  - Closed diagonal deadlocks → 検出しない
  - Bipartite deadlocks → 検出しない
  - Deadlocks due to frozen boxes → 検出しない

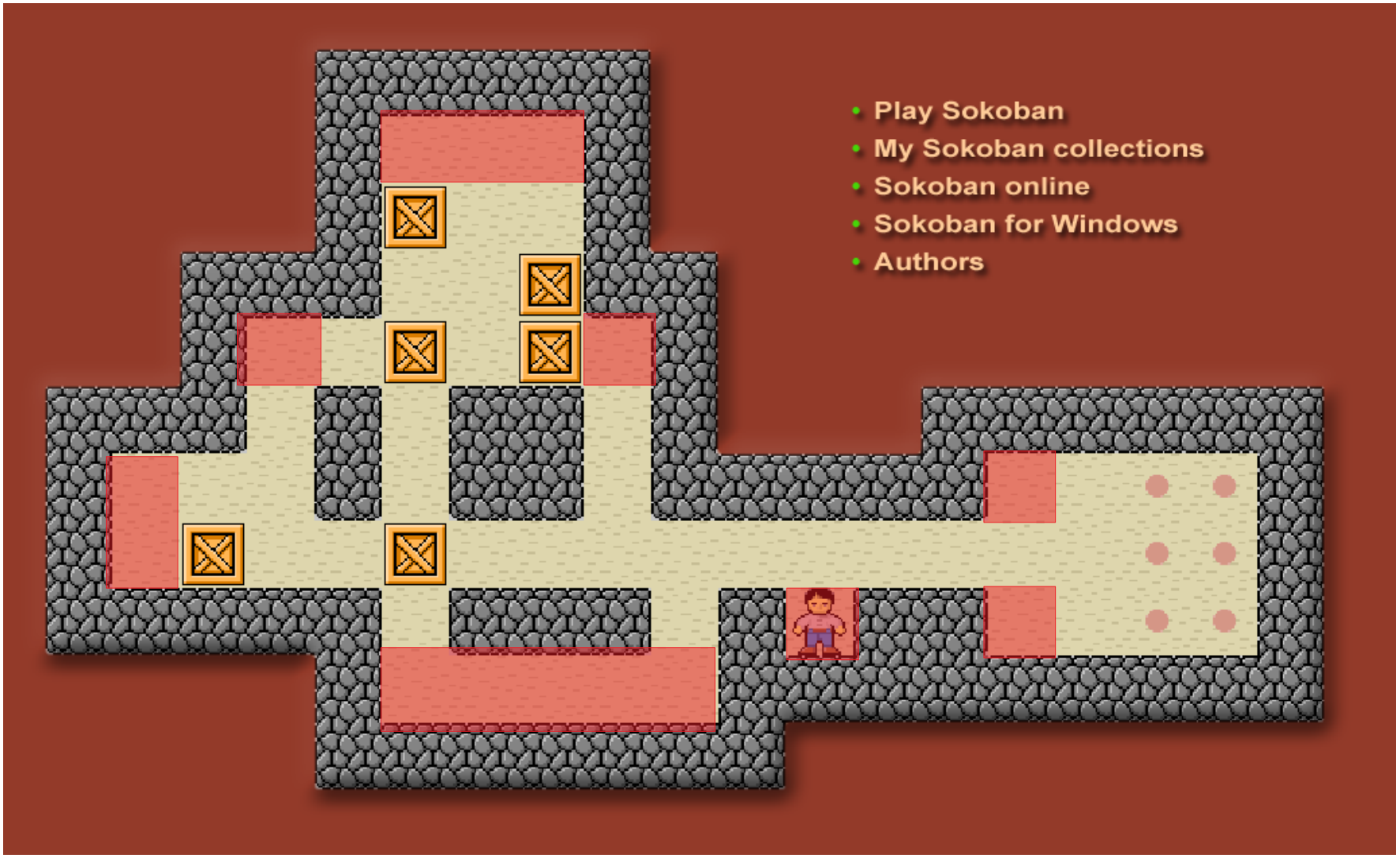
マップの静的解析

- 箱が入ってはいけない場所を探索前に探す
  - Dead square deadlock の検出

マップの静的解析


Play Sokoban
My Sokoban collections
Sokoban online
Sokoban for Windows
Authors

マップの静的解析

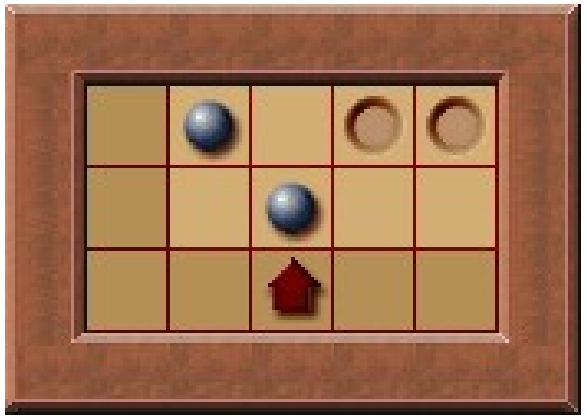

Play Sokoban
My Sokoban collections
Sokoban online
Sokoban for Windows
Authors

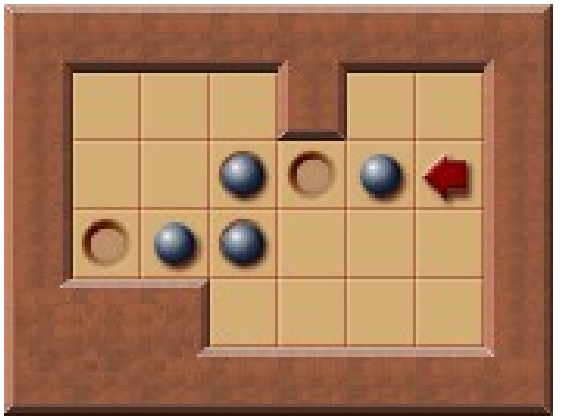
マップの静的解析

1. マップから箱をすべて削除する
2. まだチェックしていないゴールに箱を置く
3. 箱を引いて到達できる箇所を探索
4. チェックしていないゴールがあれば 2 に戻る
5. 箱が入ってはいけない場所が分かる

マップの動的解析

- 他のデッドロックは箱が関与するので動的に調べる
  - **Freeze deadlock** の検出
  - 箱が移動する度に調べる





マップの動的解析

次のいずれかに当てはまる箱は軸に対して移動不能

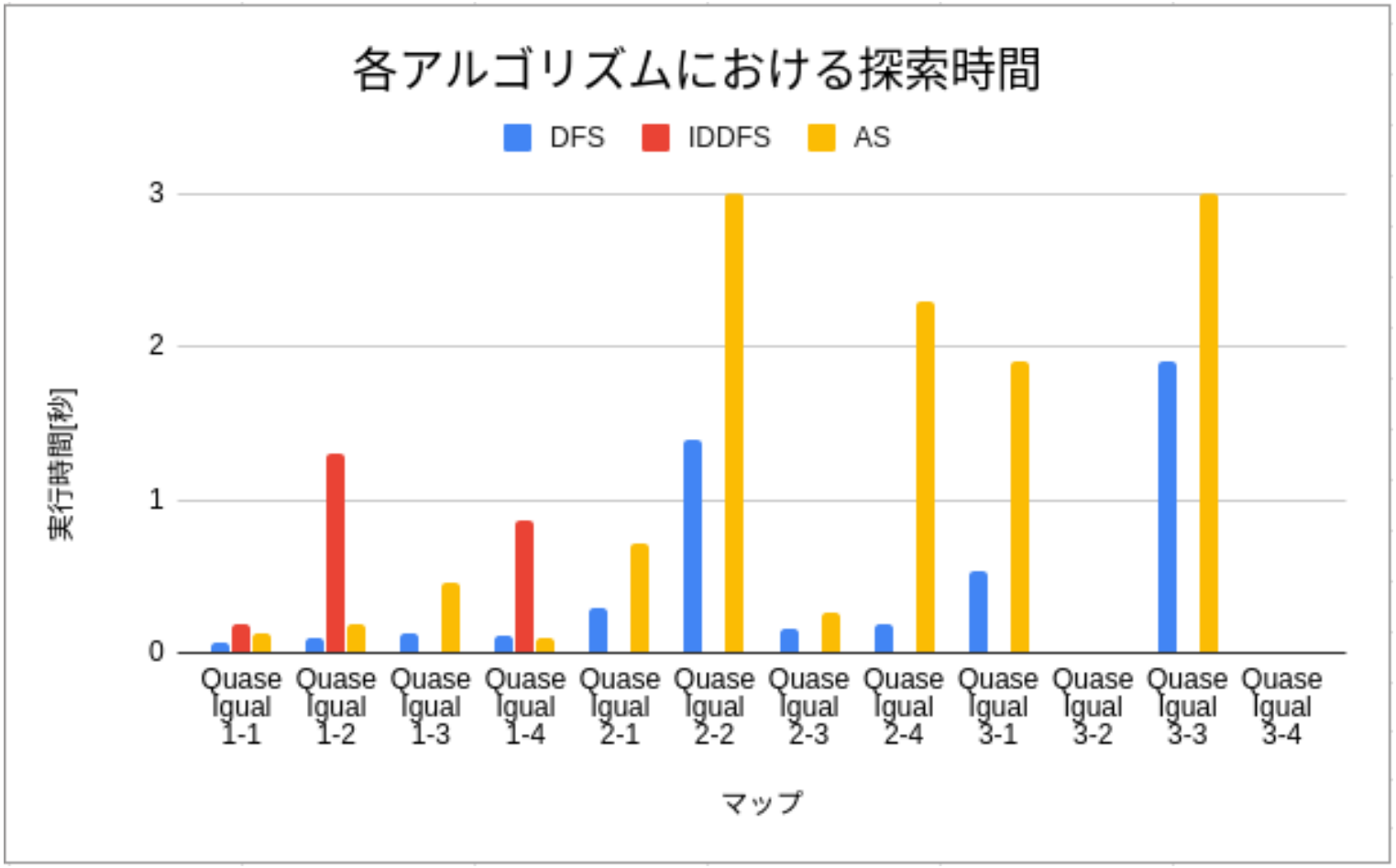
1. [左右 / 上下] どちらかに壁がある
2. [左右 / 上下] 両方に dead square がある
3. [左右 / 上下] どちらかに移動不能な箱がある

縦横両方について移動不能なら、箱は移動不能

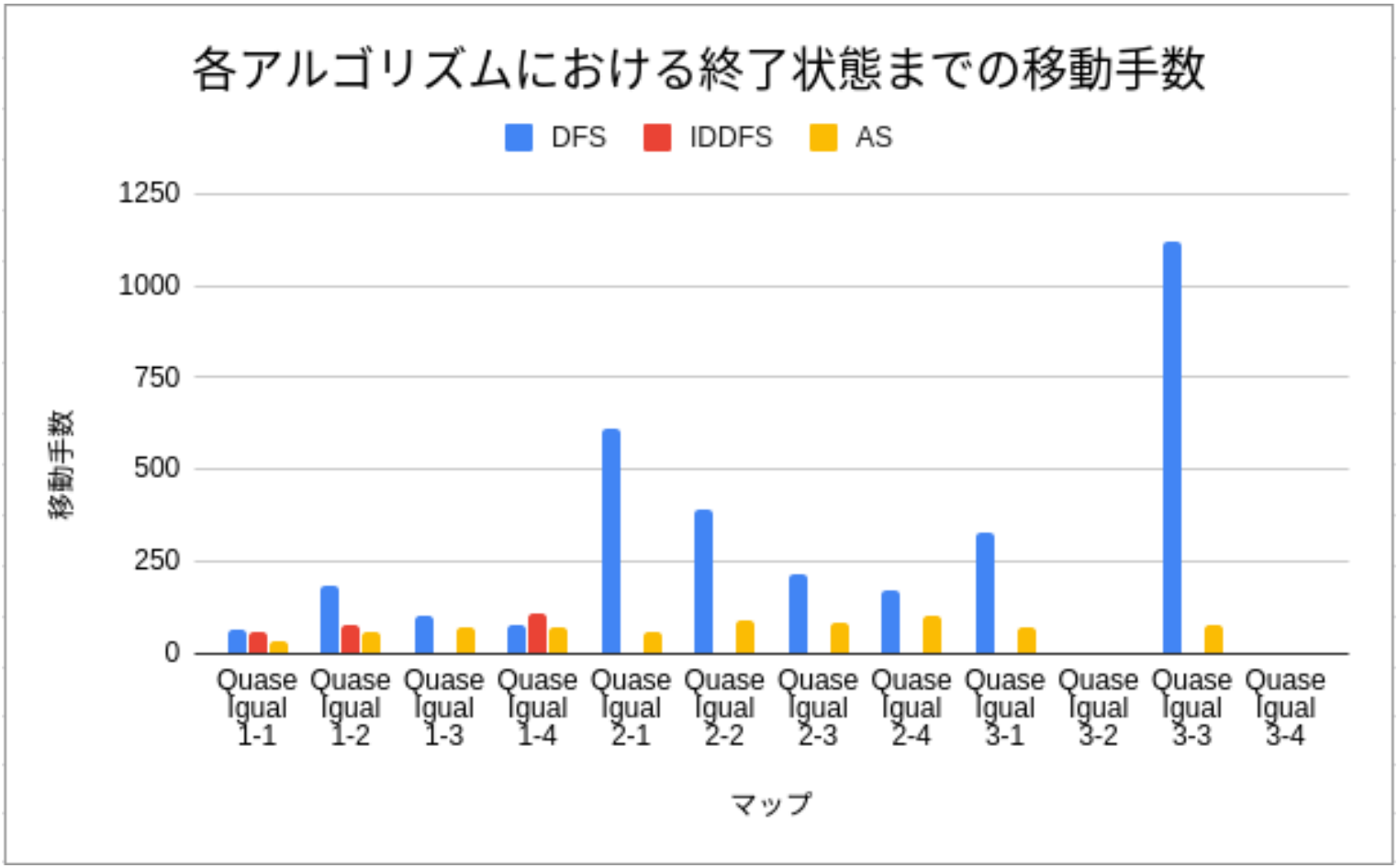
時間の都合上 3 は浅い検知しかしていない

デモ

解を見つけるまでの時間



見つかった解の手数

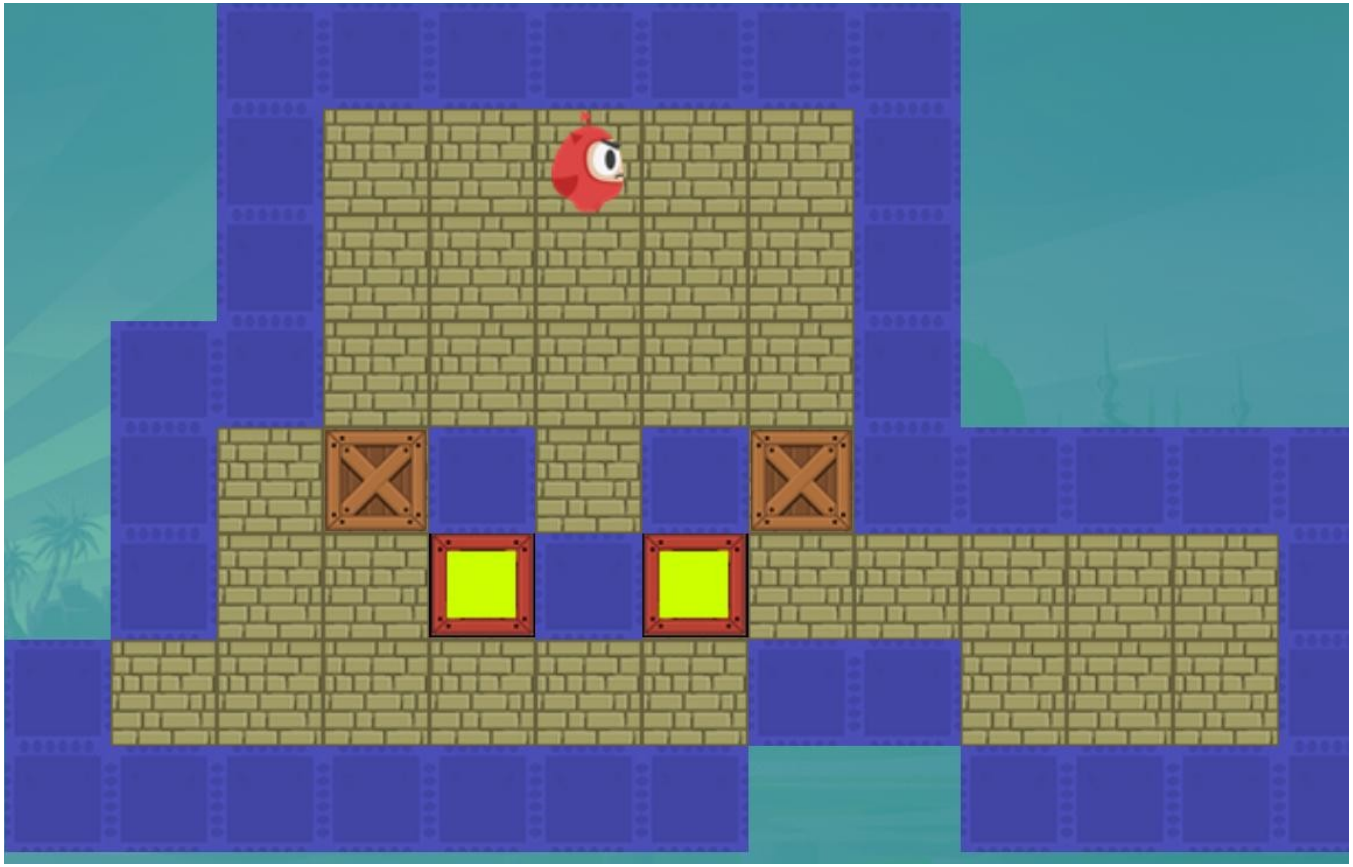

各アルゴリズムにおける終了状態までの移動手数
DFS
IDDFS
AS
1250
1000
750
500
250
0
移動手数
Quase Igual 1-1
Quase Igual 1-2
Quase Igual 1-3
Quase Igual 1-4
Quase Igual 2-1
Quase Igual 2-2
Quase Igual 2-3
Quase Igual 2-4
Quase Igual 3-1
Quase Igual 3-2
Quase Igual 3-3
Quase Igual 3-4
マップ

実行結果の考察

- DFS 速い
- A* かしこい
- 探索深さの幅が大きいので IDDFS は向いてない
- 小さいマップは難しくても解ける
- 大きいマップは易しくても解けない
- 箱が多いほど探索に時間がかかる

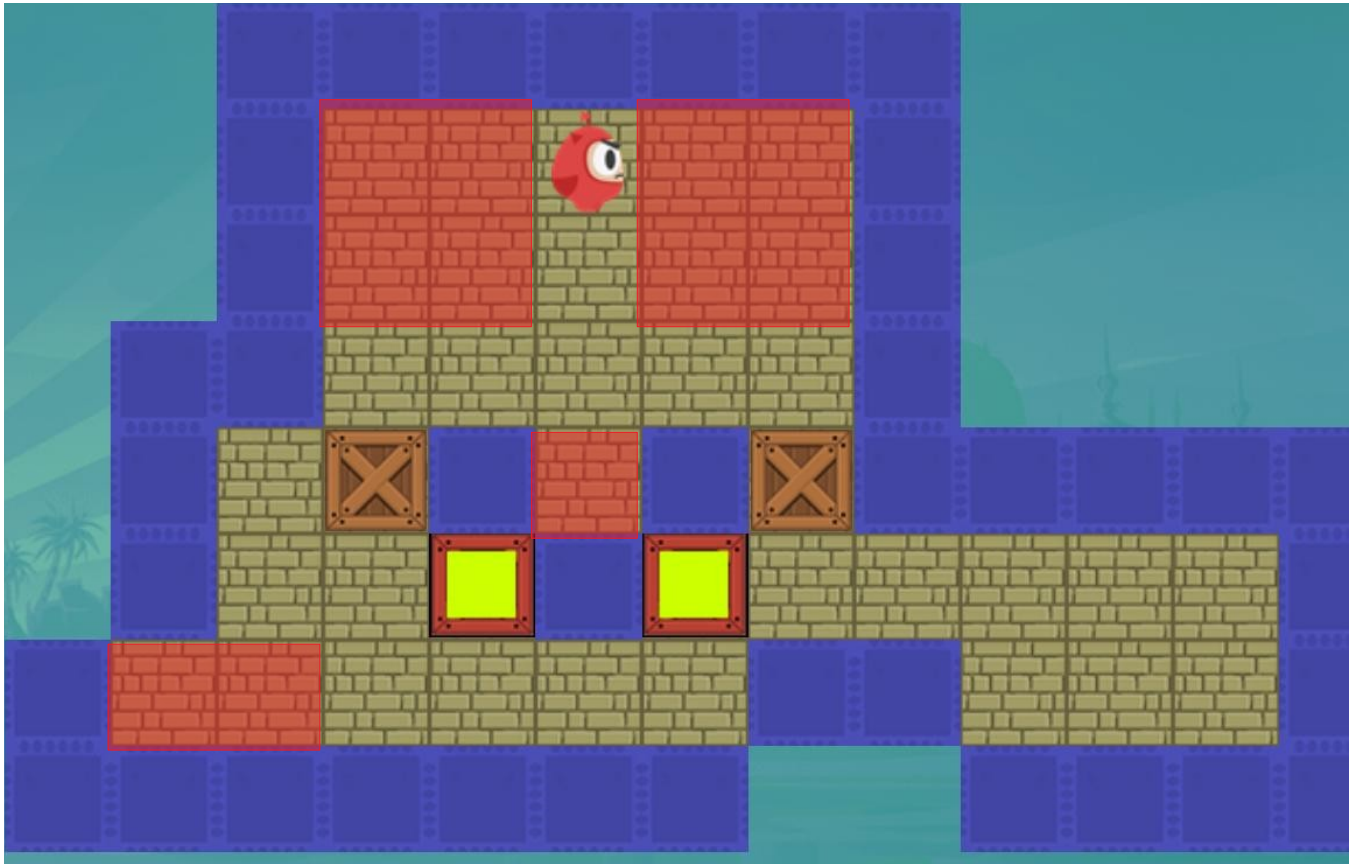
高速化に関する考察

- 広い空間を等価な狭い空間に置き換えたい



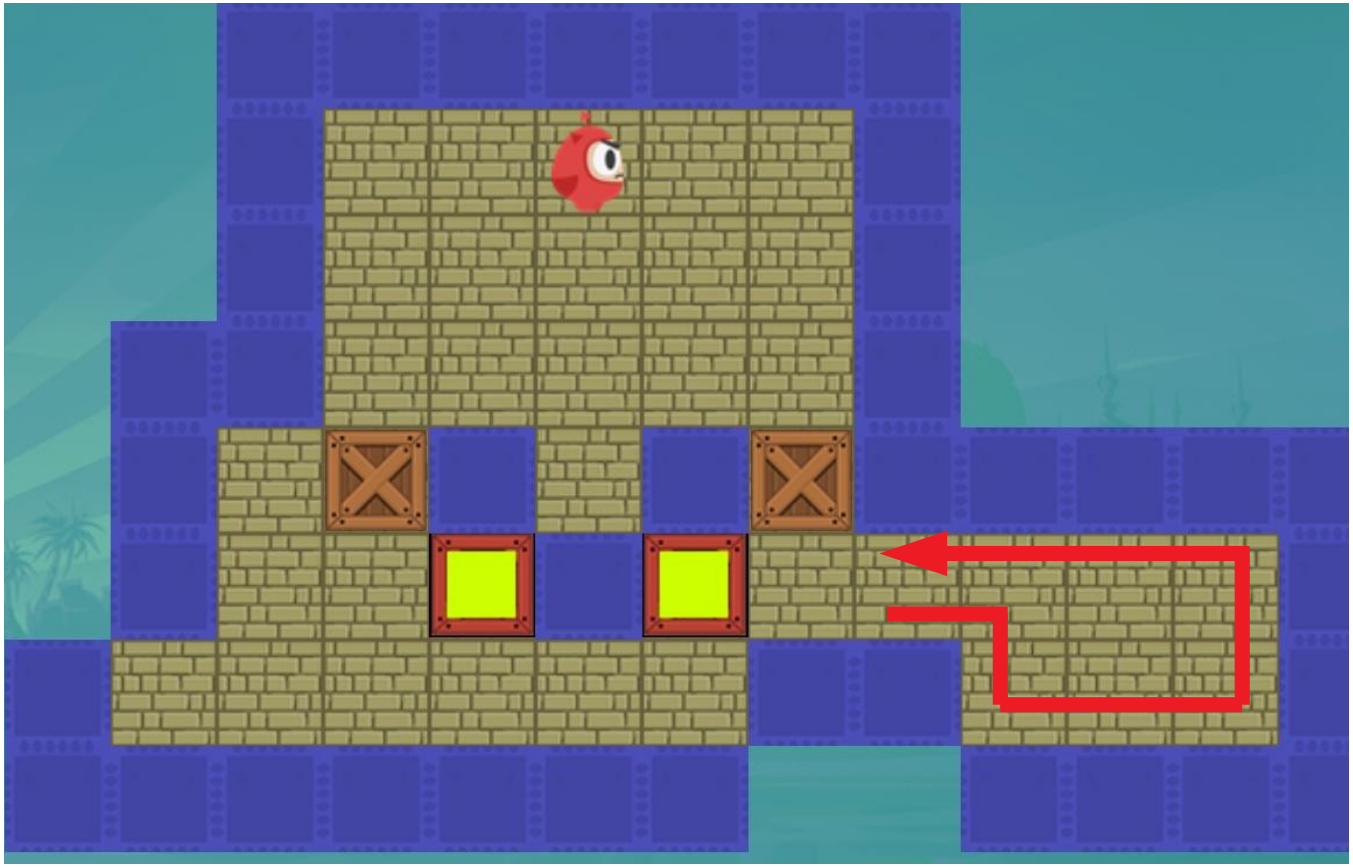
高速化に関する考察

- 広い空間を等価な狭い空間に置き換えたい



高速化に関する考察

- 手数のかかる場所にパターンを用意したい



ソースコード

https://github.com/ptr-yudai/titech-sokoban