関数・論理型プログラミング実験
論理型演習第3回
（通算第13回）

松田 一孝

TA: 武田広太郎 寺尾拓

これまでの流れ

- 第1回 (6/24)
  - Prologの使い方
- 第2回 (7/1)
  - Prologの評価メカニズム
- 第3回 (7/8)
  - いろいろな探索
- 第4回 (7/15)
  - 関数論理型言語Curry

今日の話

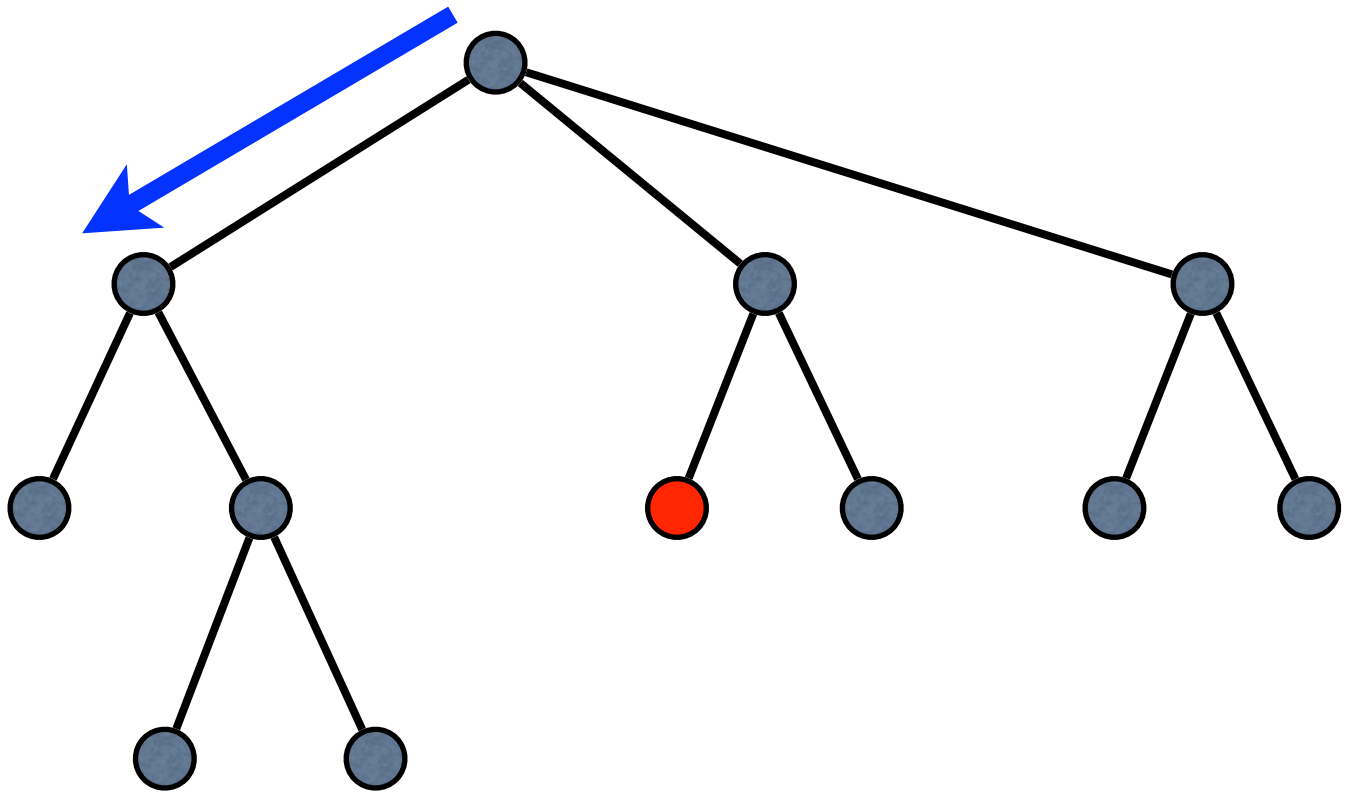
- いろいろな探索手法を学ぶ
  - 深さ優先探索（Depth-First Search）
  - 幅優先探索（Breadth-First Search）
  - 反復深化（Iterative Deepening）
  - A*

ねらい

- Prologの探索は深さ優先
  - 特徴を知ることでより活用可
- いくつかのシステムでは探索木を取り出し，操作できる
  - 例
    - HaskellのMonadPlus
    - CurryのgetSearchTree
      - getSearchTree :: a -> IO (SearchTree a)
  - これらをさらに活用できるように
- そうでなくとも探索は重要

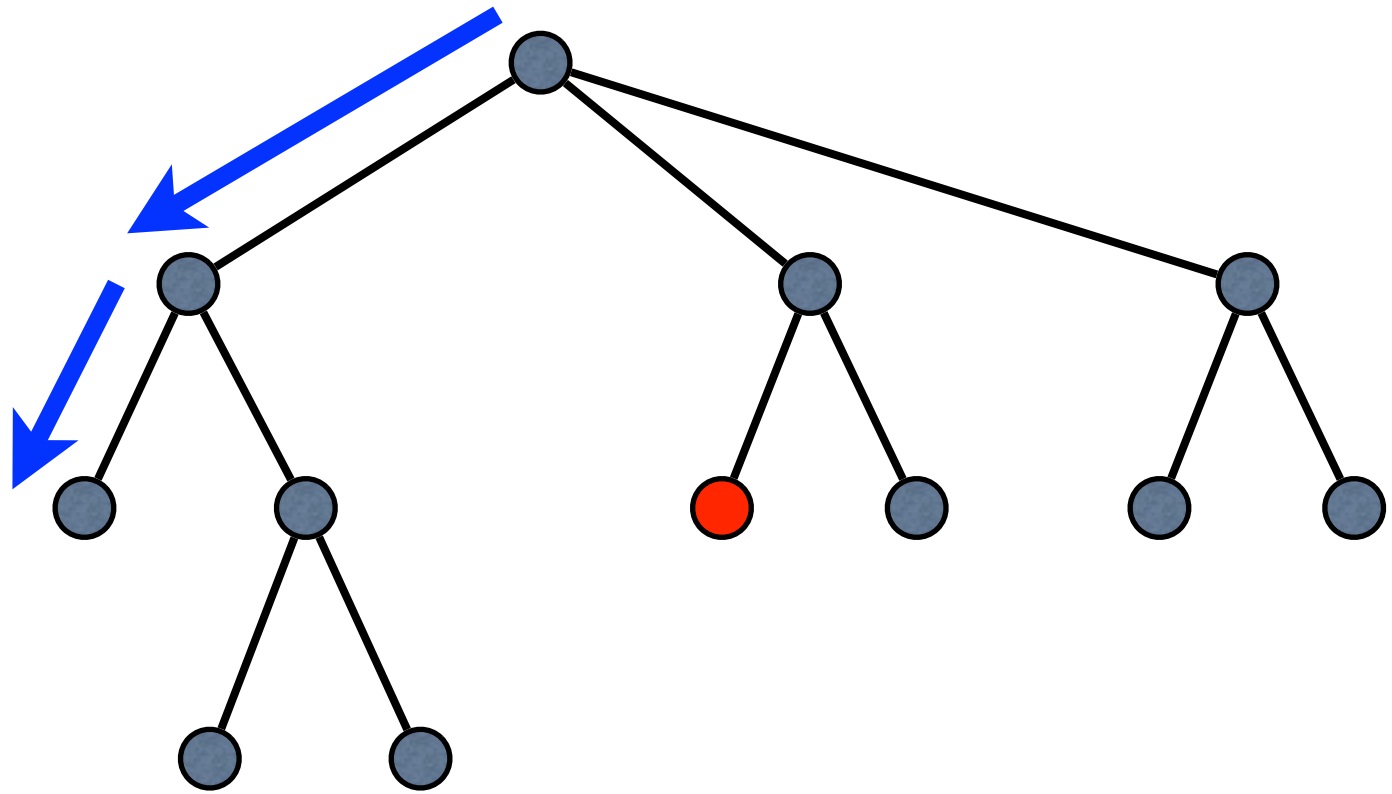
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



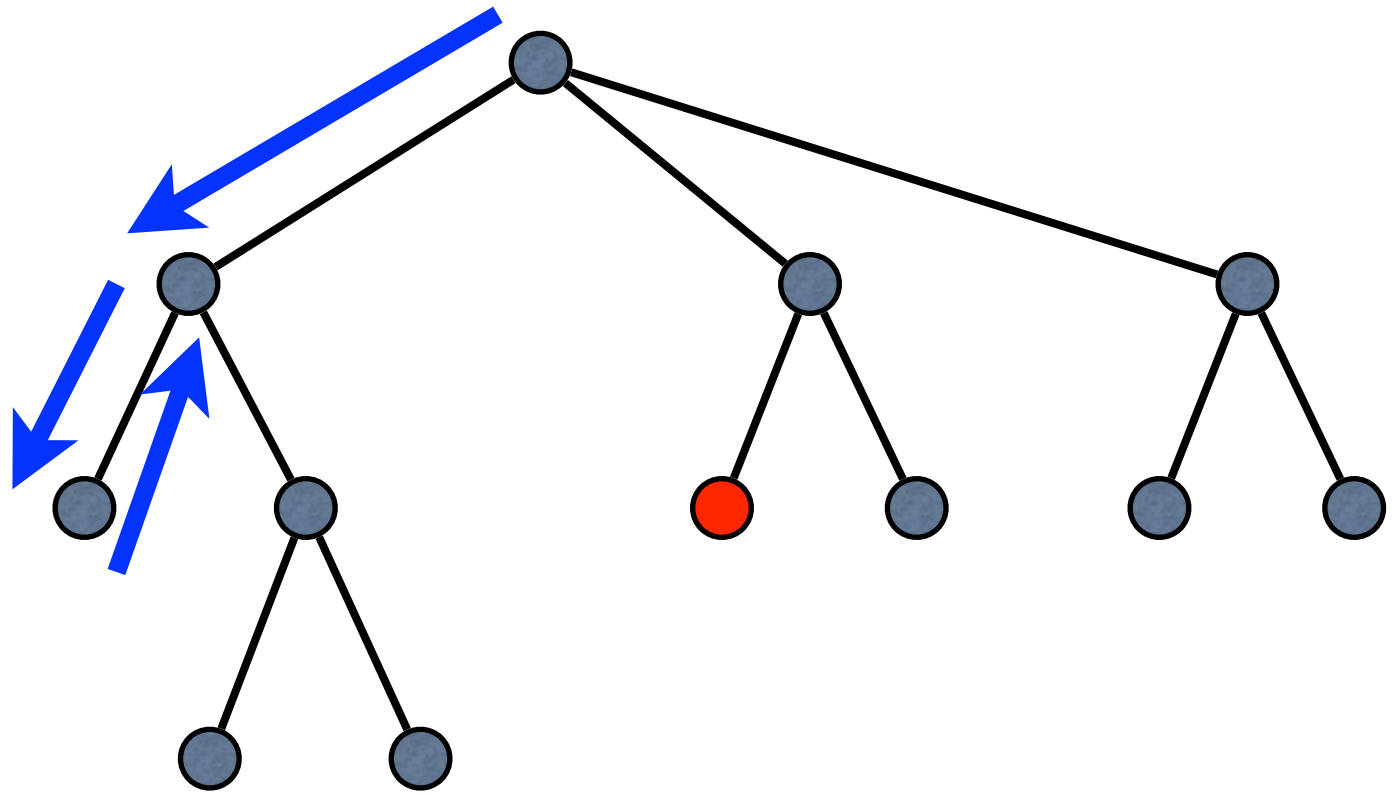
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



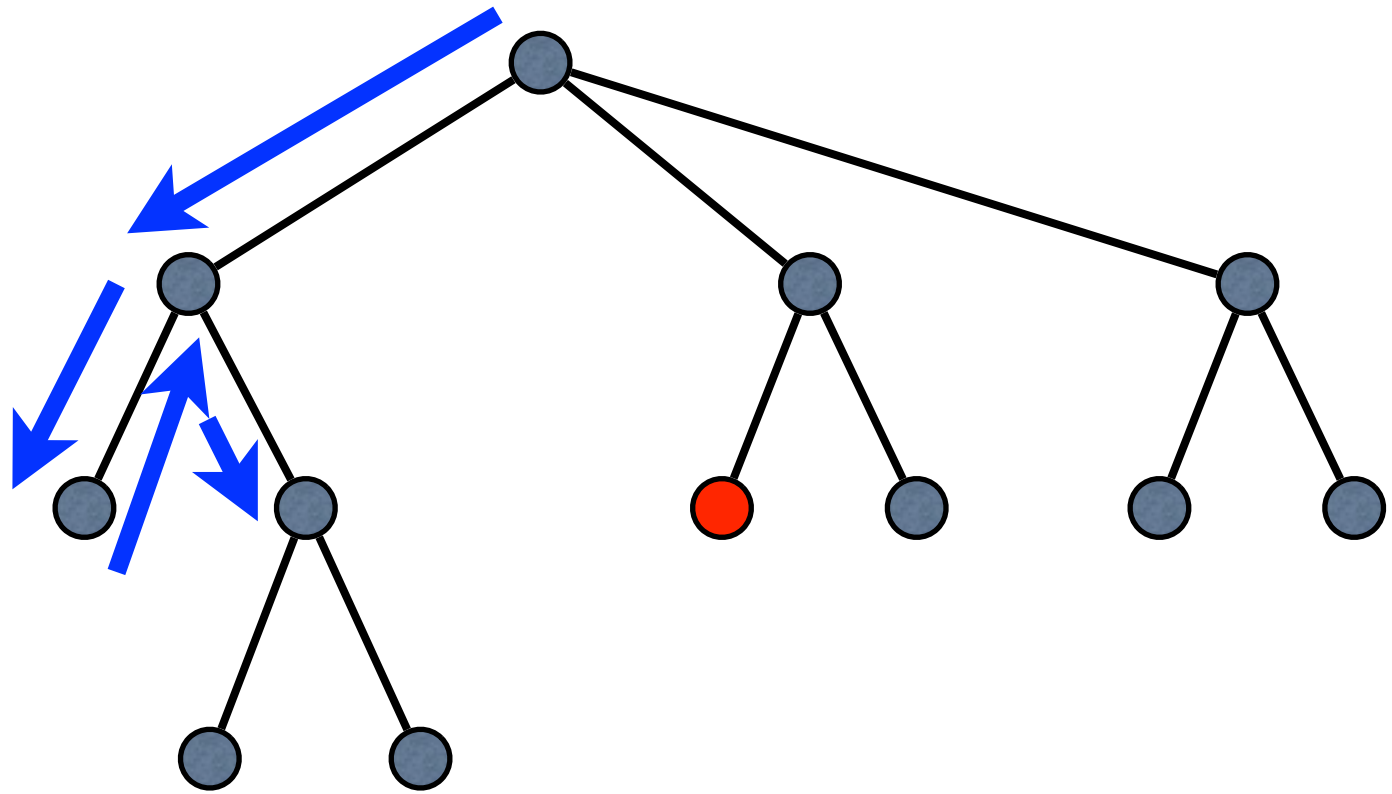
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



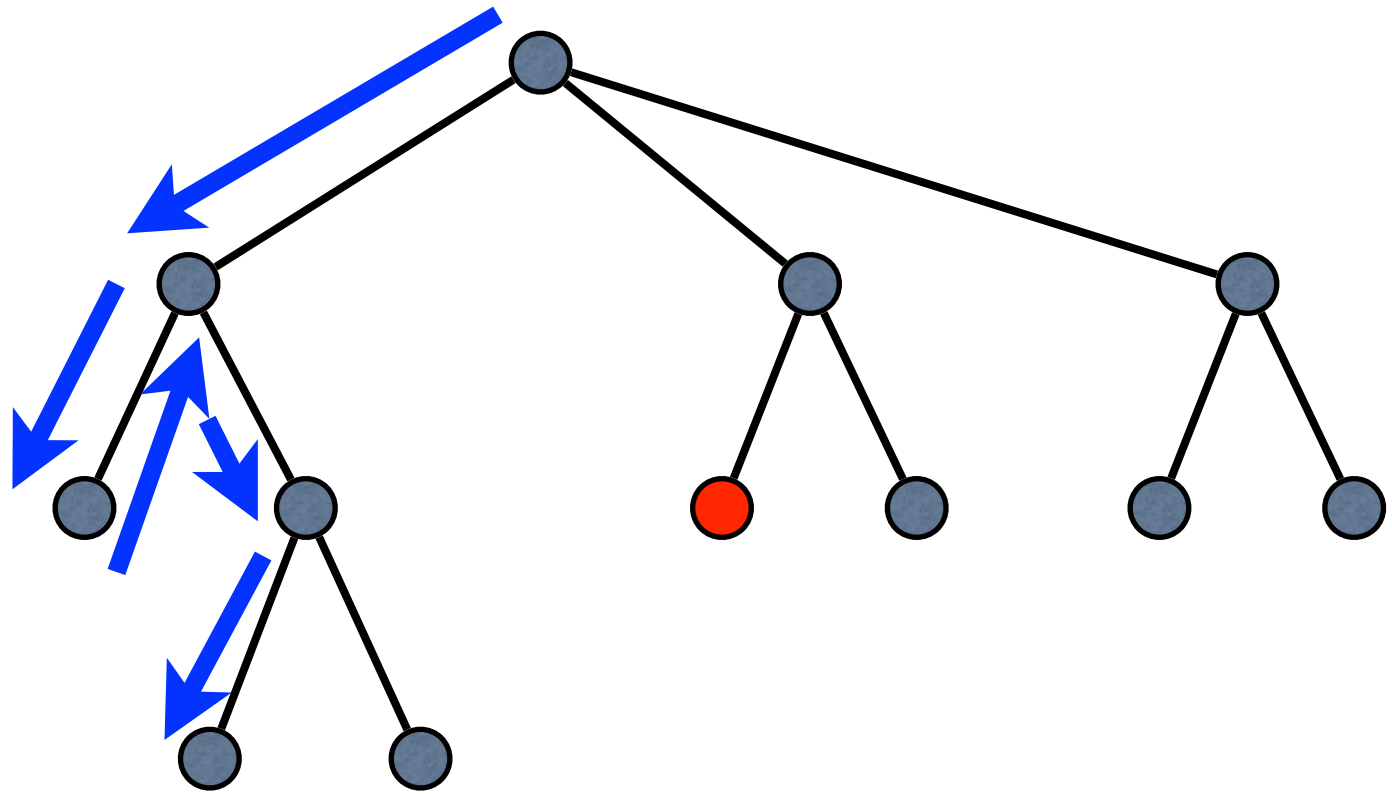
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



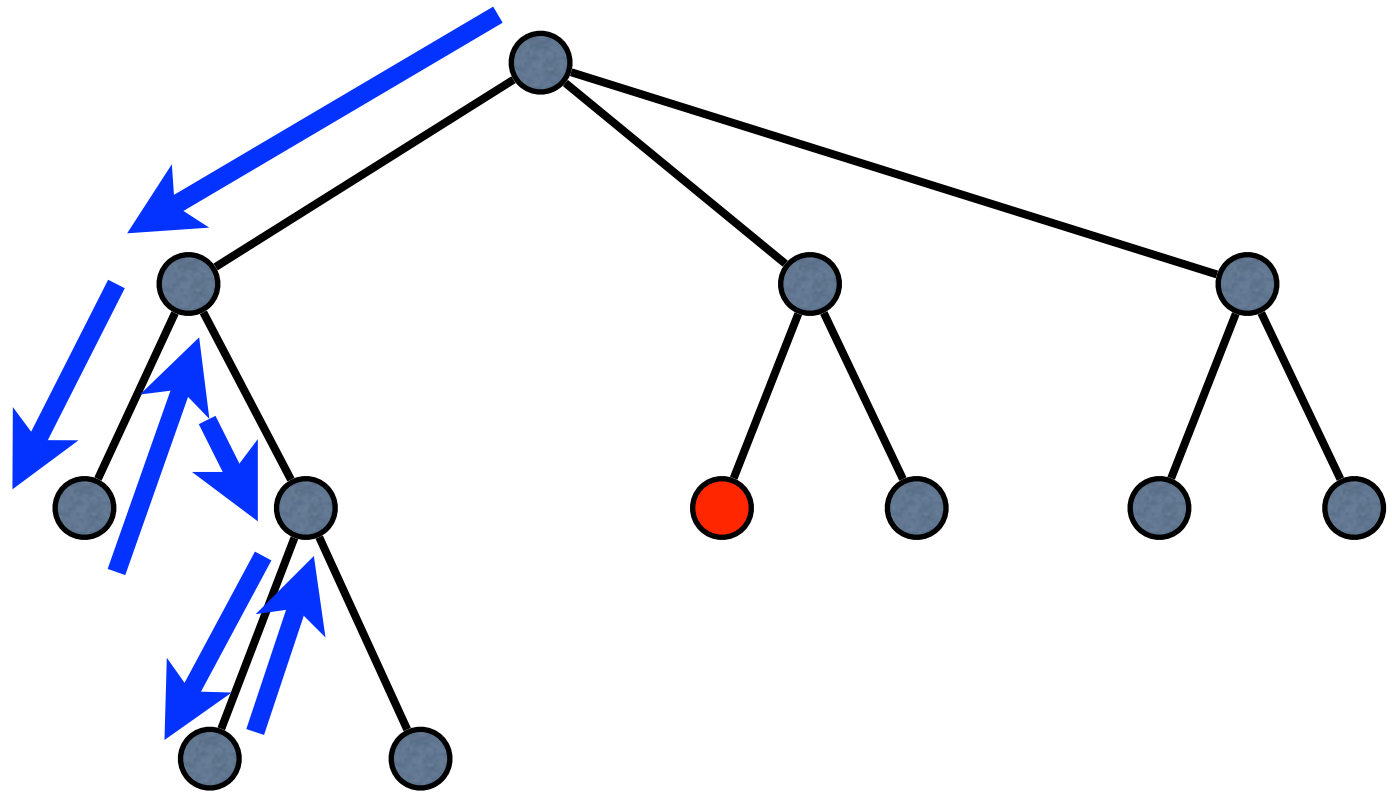
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



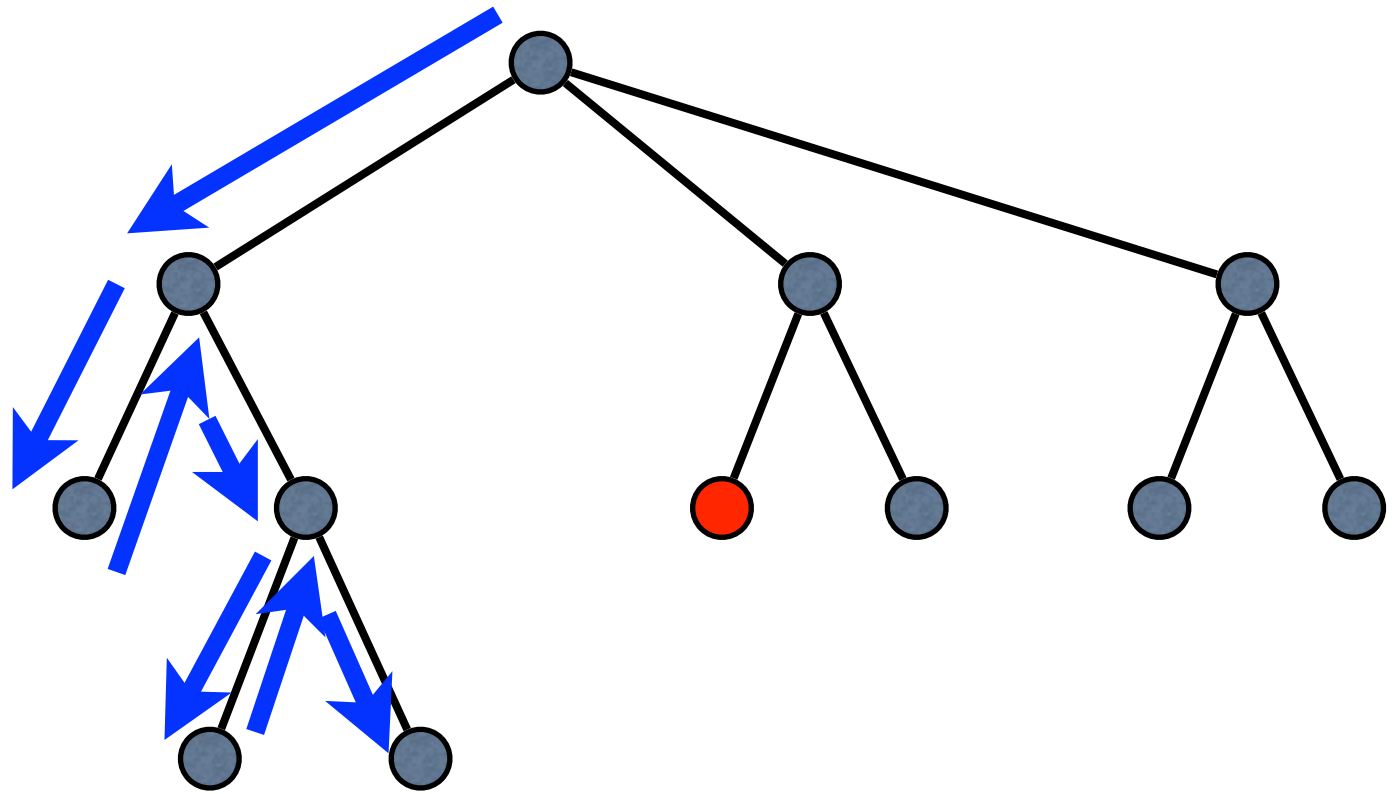
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



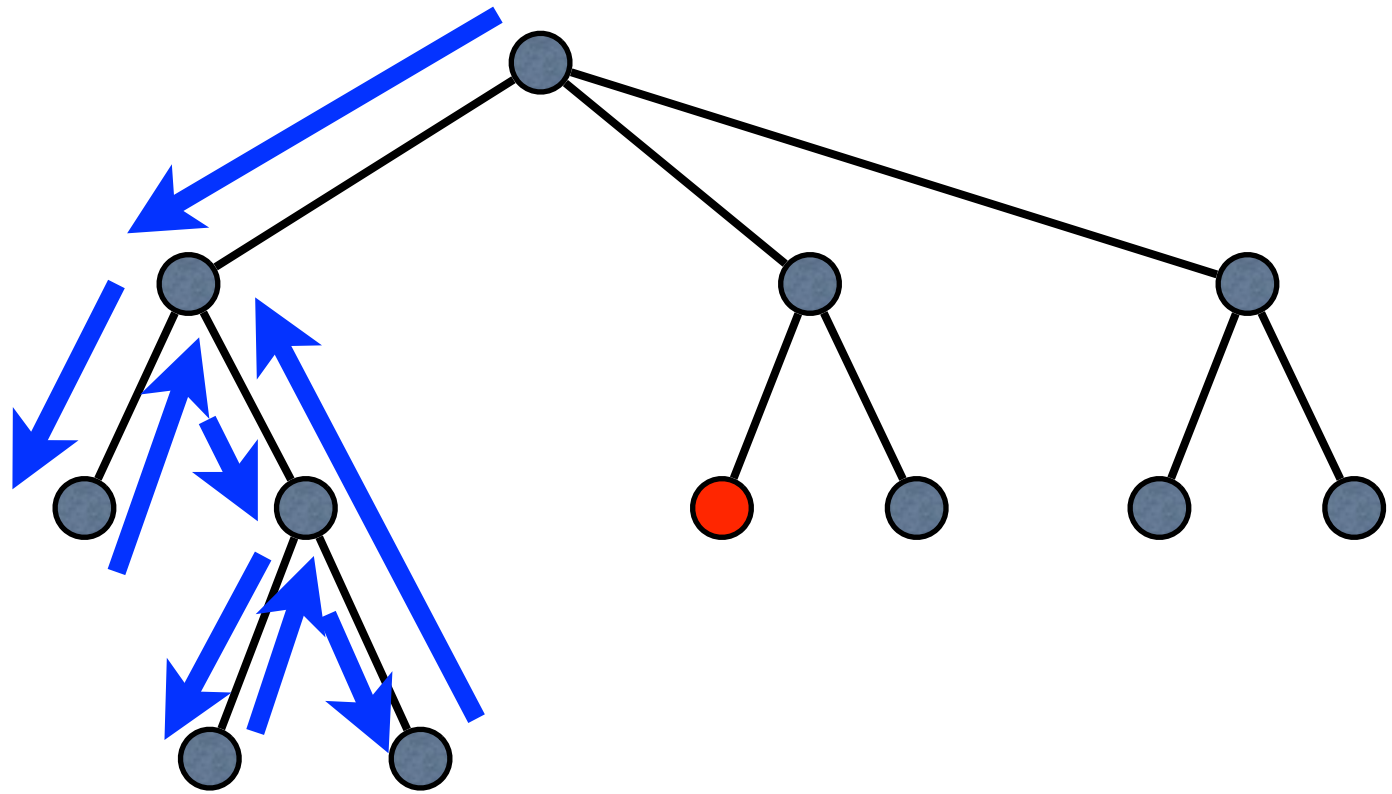
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



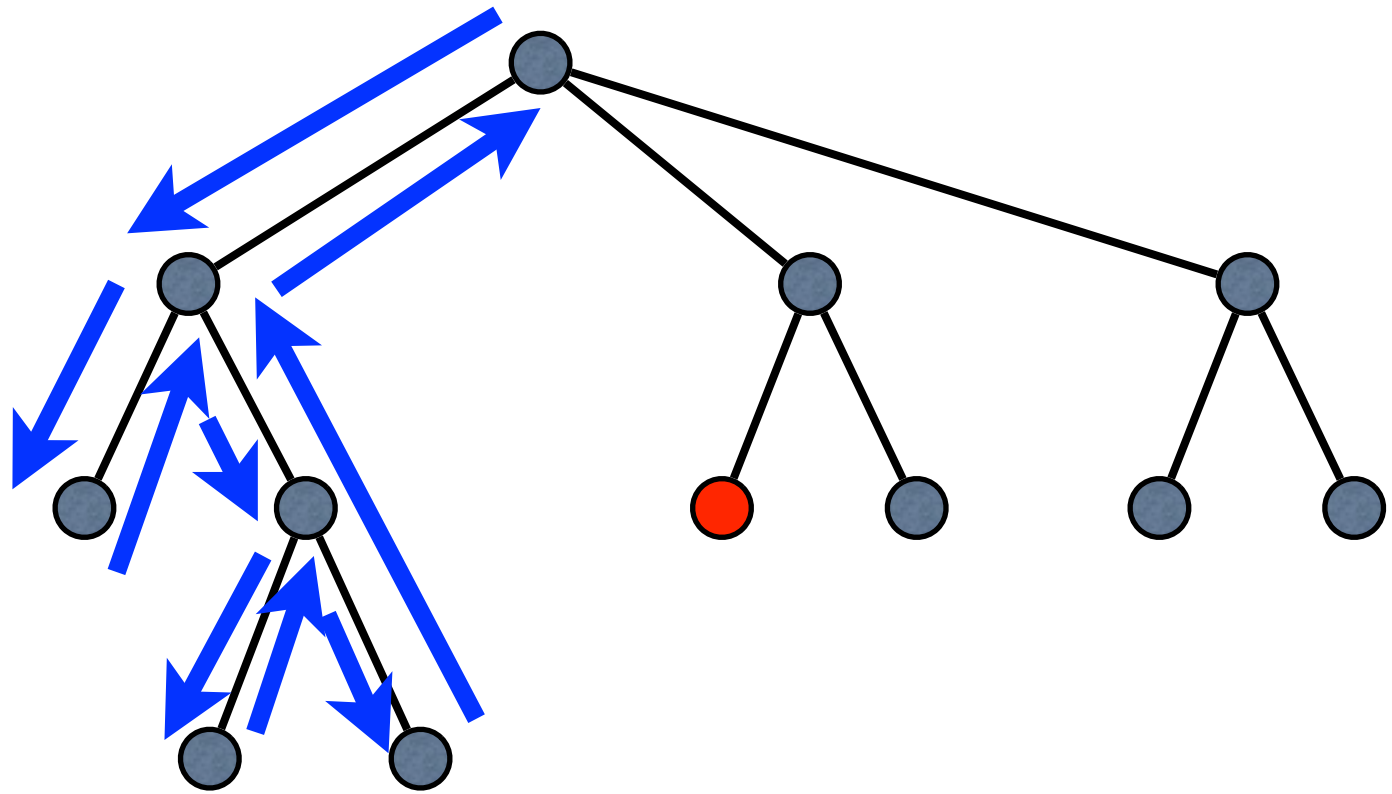
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



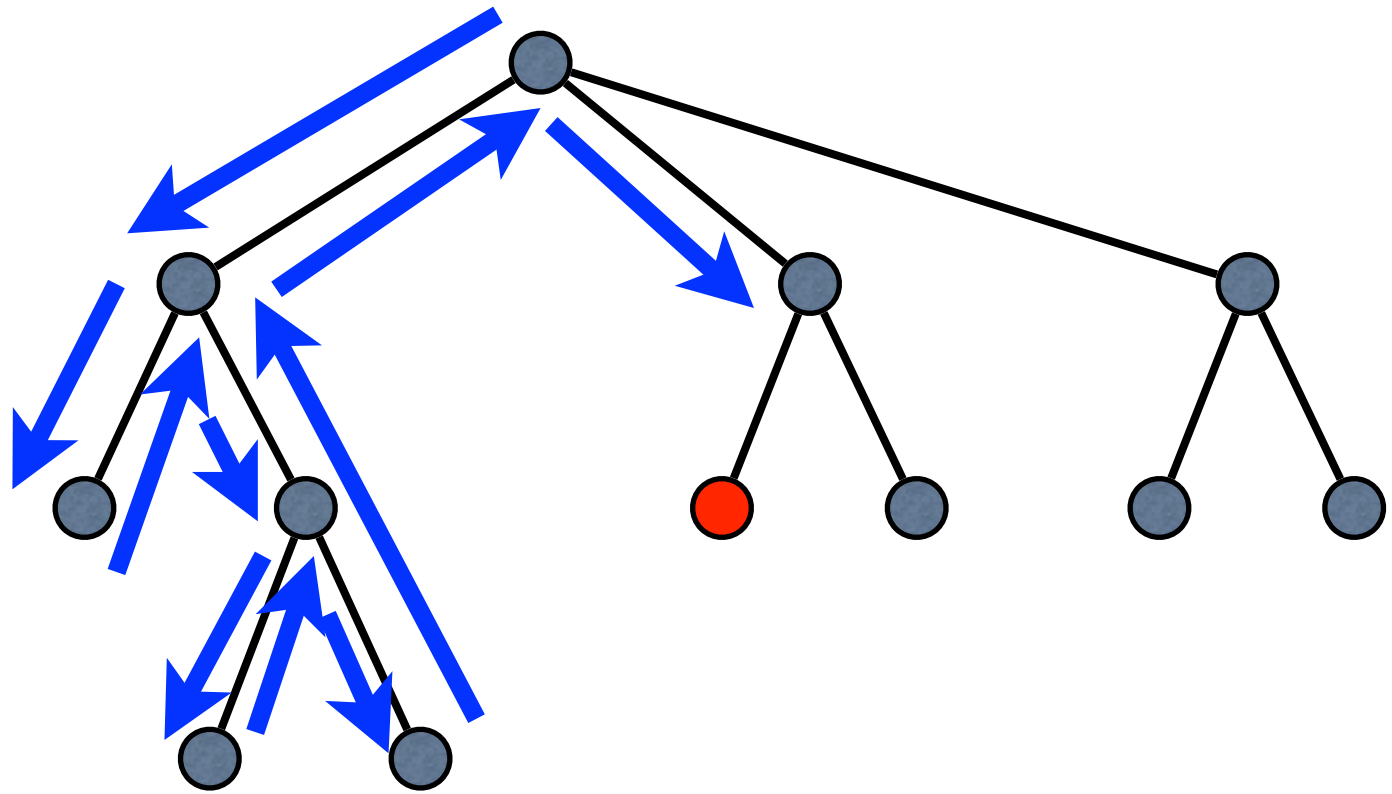
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



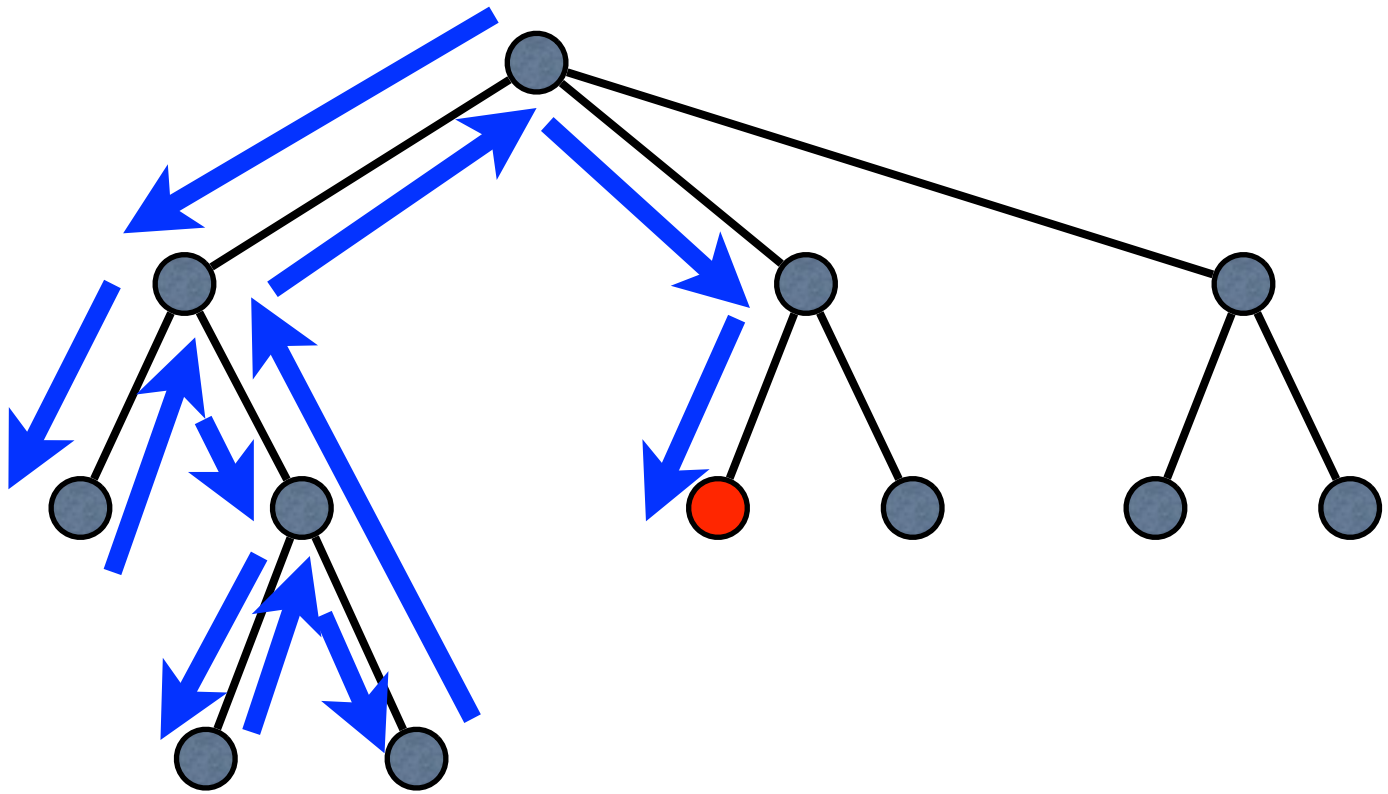
深さ優先探索

- 深く掘る ＞ 別の可能性を試す



深さ優先探索

- 深く掘る ＞ 別の可能性を試す



Haskellでの実装例

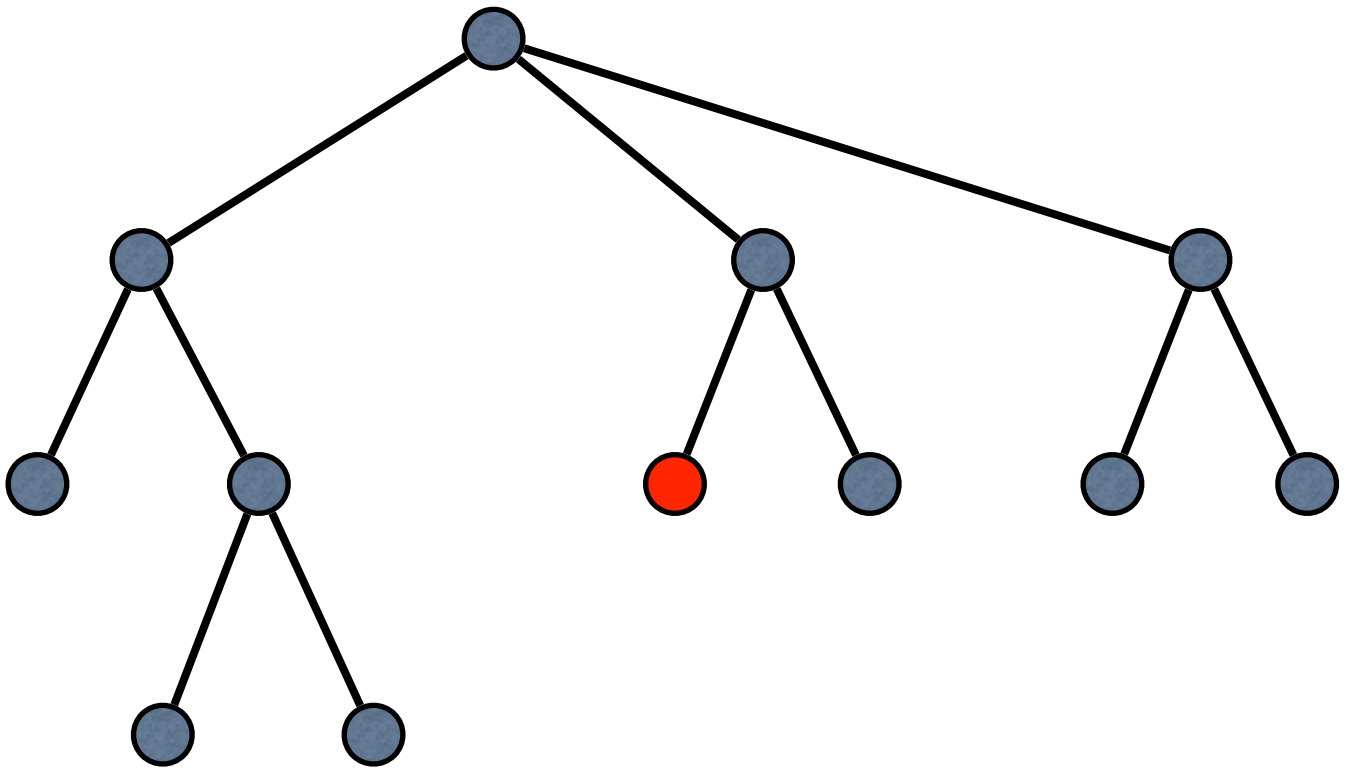
```
data SearchT a
    = SNone | SUnit a
    | SOr (SearchT a) (SearchT a)

dfs :: SearchT a -> Maybe a
dfs t = go [t]
  where
    go []             = Nothing
    go (SNone:x)      = go x
    go (SUnit a:x)    = Just a
    go (SOr l r:x)    = go (l:r:x)
```

利点・欠点

- 利点
  - 実装が容易
    - スタック
      - 関数呼出のスタックを利用可
  - 空間効率がよい
    - 「深さ」分だけあればよい
- 欠点
  - あるはずの解が発見できない
    - 無限に深い部分木（前回参照）
  - 「最短で到達」する解を探すには不向き

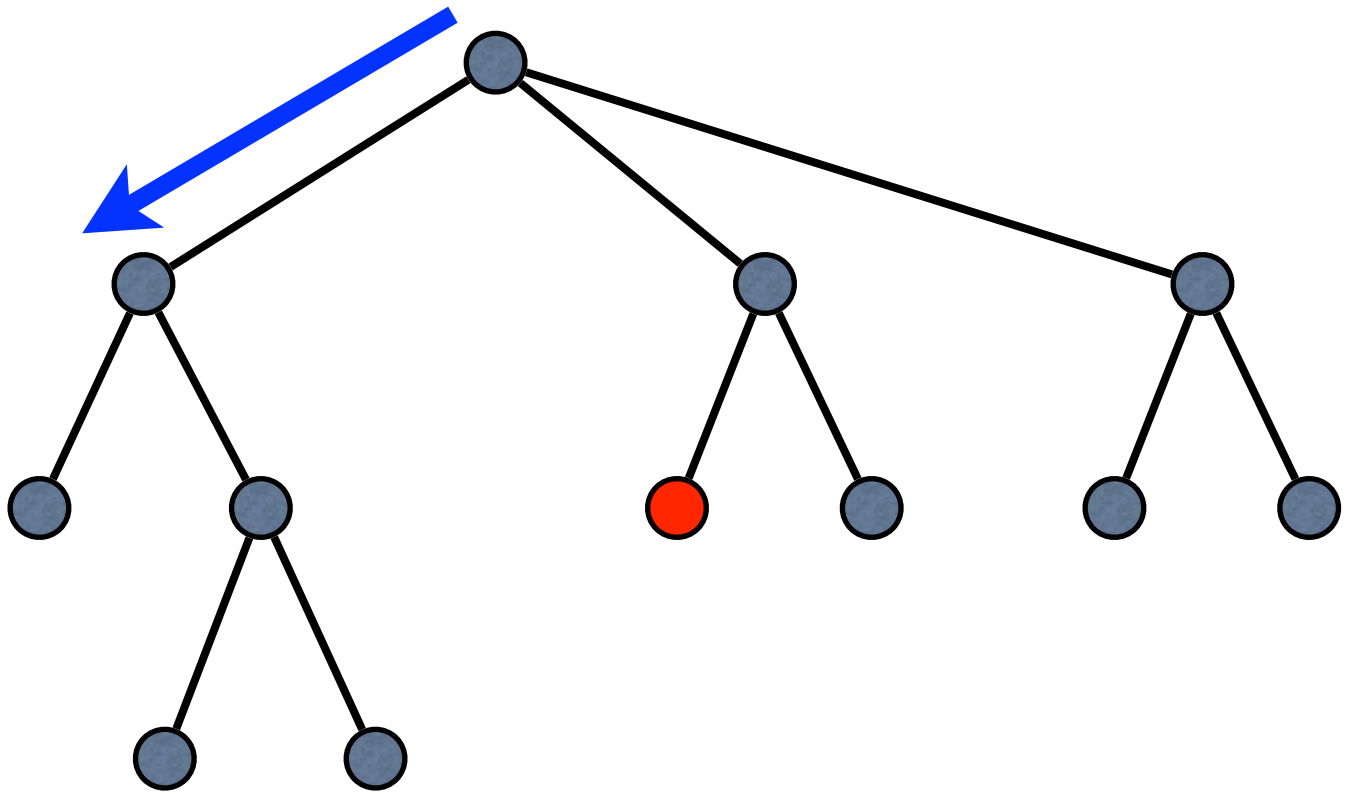
幅優先探索

- 別の可能性を試す ＞ 深く掘る



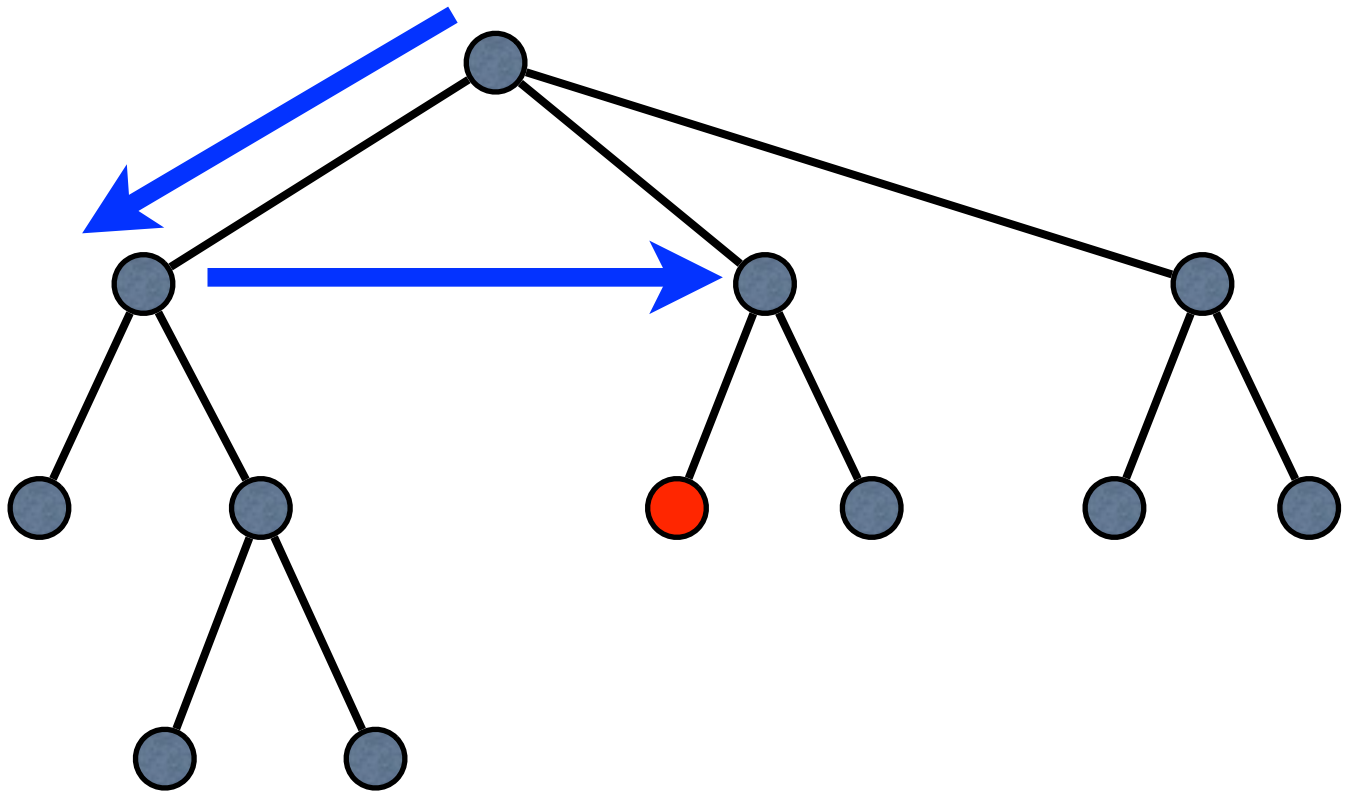
幅優先探索

- 別の可能性を試す ＞ 深く掘る



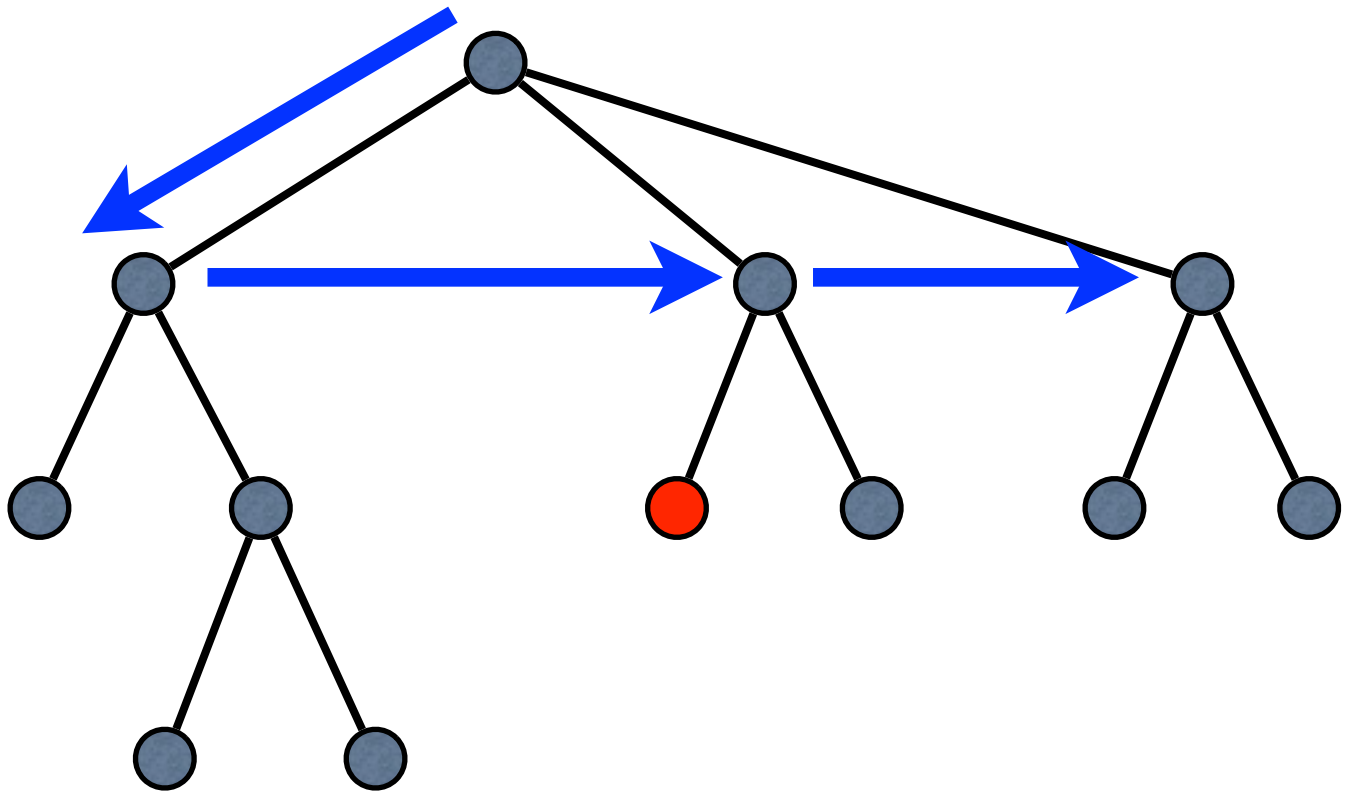
幅優先探索

- 別の可能性を試す ＞ 深く掘る



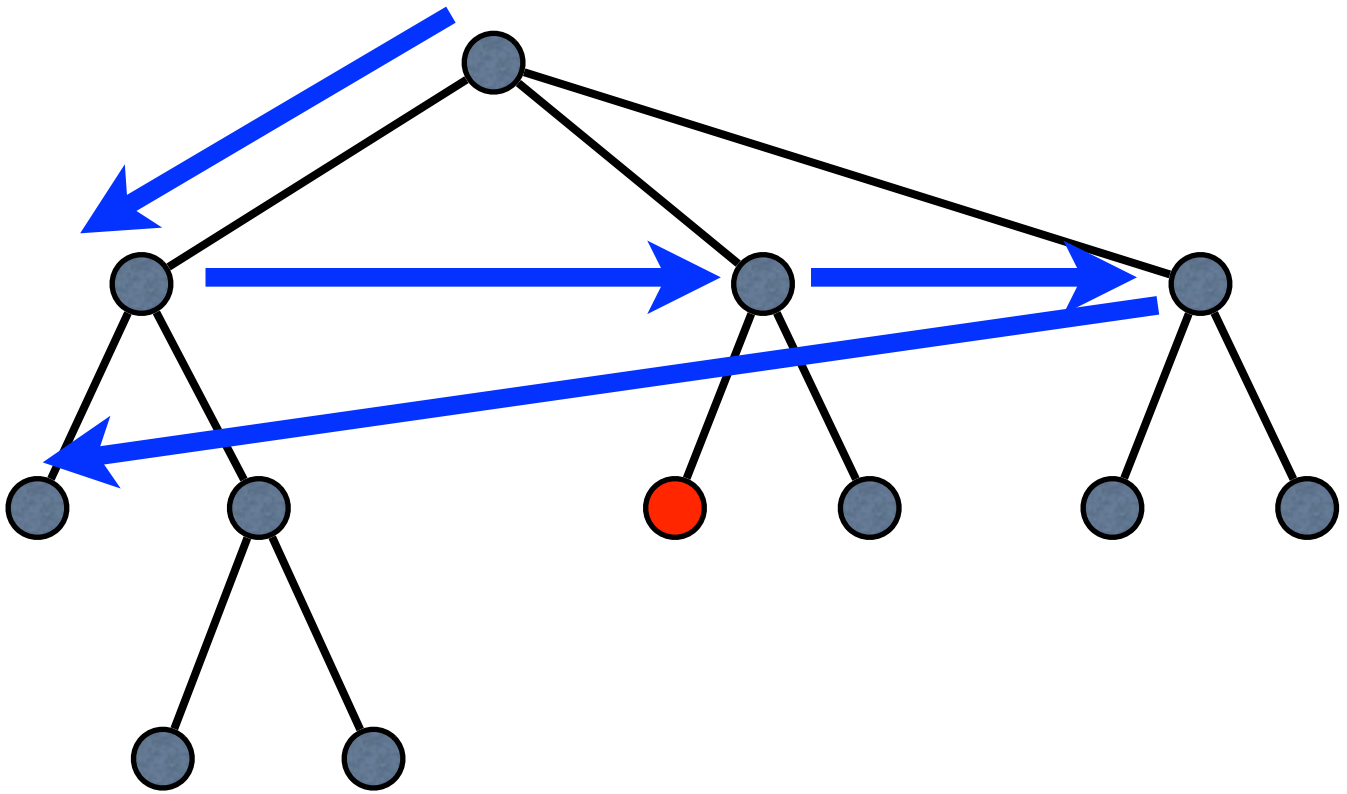
幅優先探索

- 別の可能性を試す ＞ 深く掘る



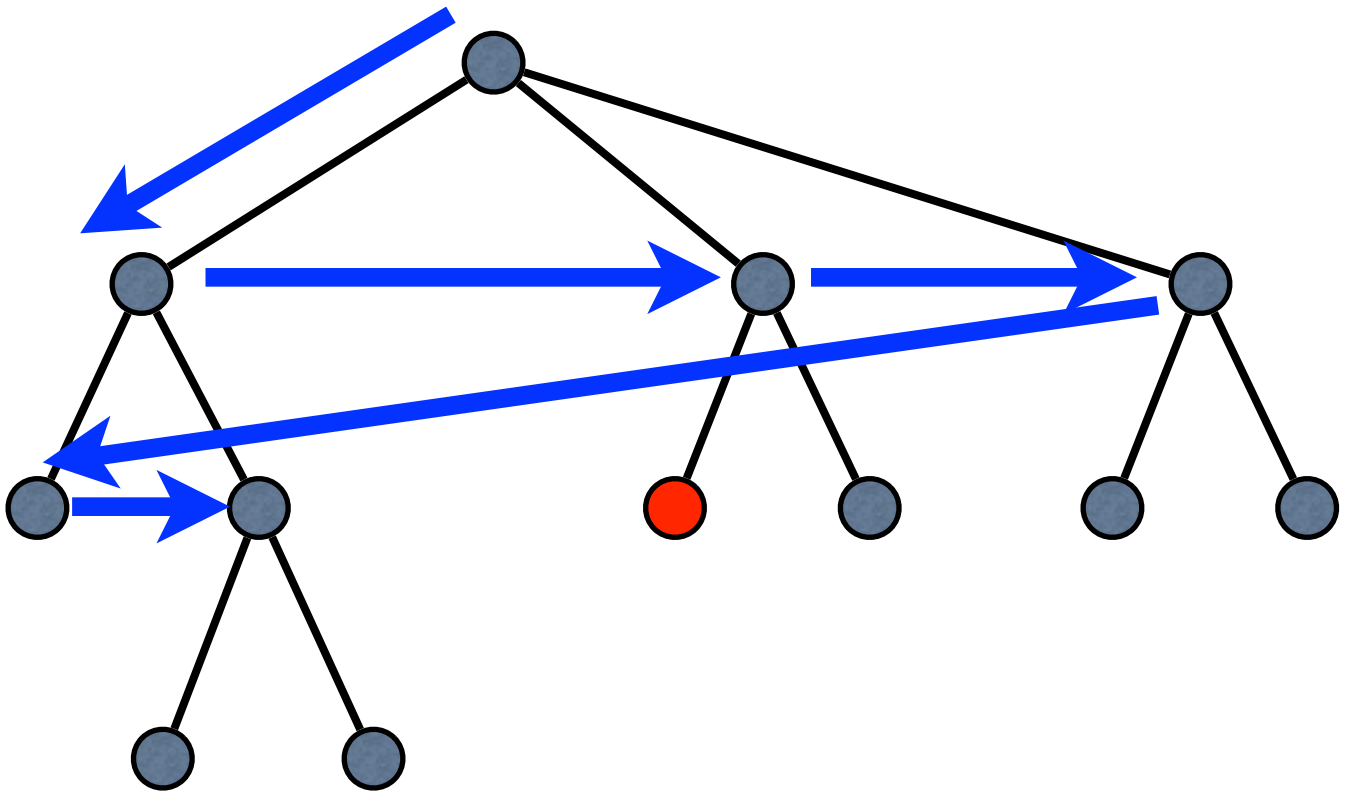
幅優先探索

- 別の可能性を試す ＞ 深く掘る



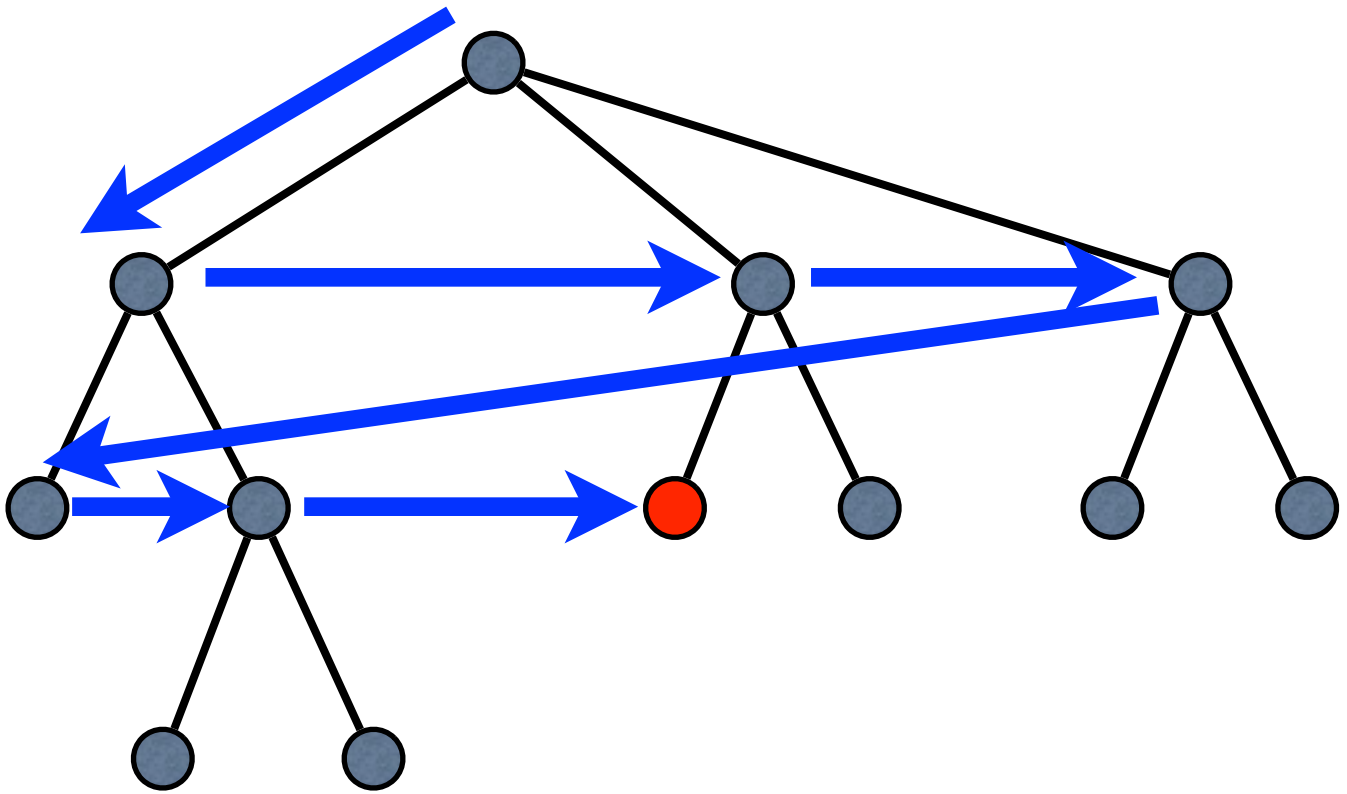
幅優先探索

- 別の可能性を試す ＞ 深く掘る



幅優先探索

- 別の可能性を試す ＞ 深く掘る



Haskellでの実装例

```
data SearchT a
    = SNone | SUnit a
    | SOr (SearchT a) (SearchT a)

bfs :: SearchT a -> Maybe a
bfs t = go [t] []
  where
    go [] []           = Nothing
    go [] y            = go (reverse y) []
    go (SNone:x)    y = go x y
    go (SUnit a:x)  y = Just a
    go (SOr l r:x)  y = go x (r:l:y)
```

遅延評価を活用した版

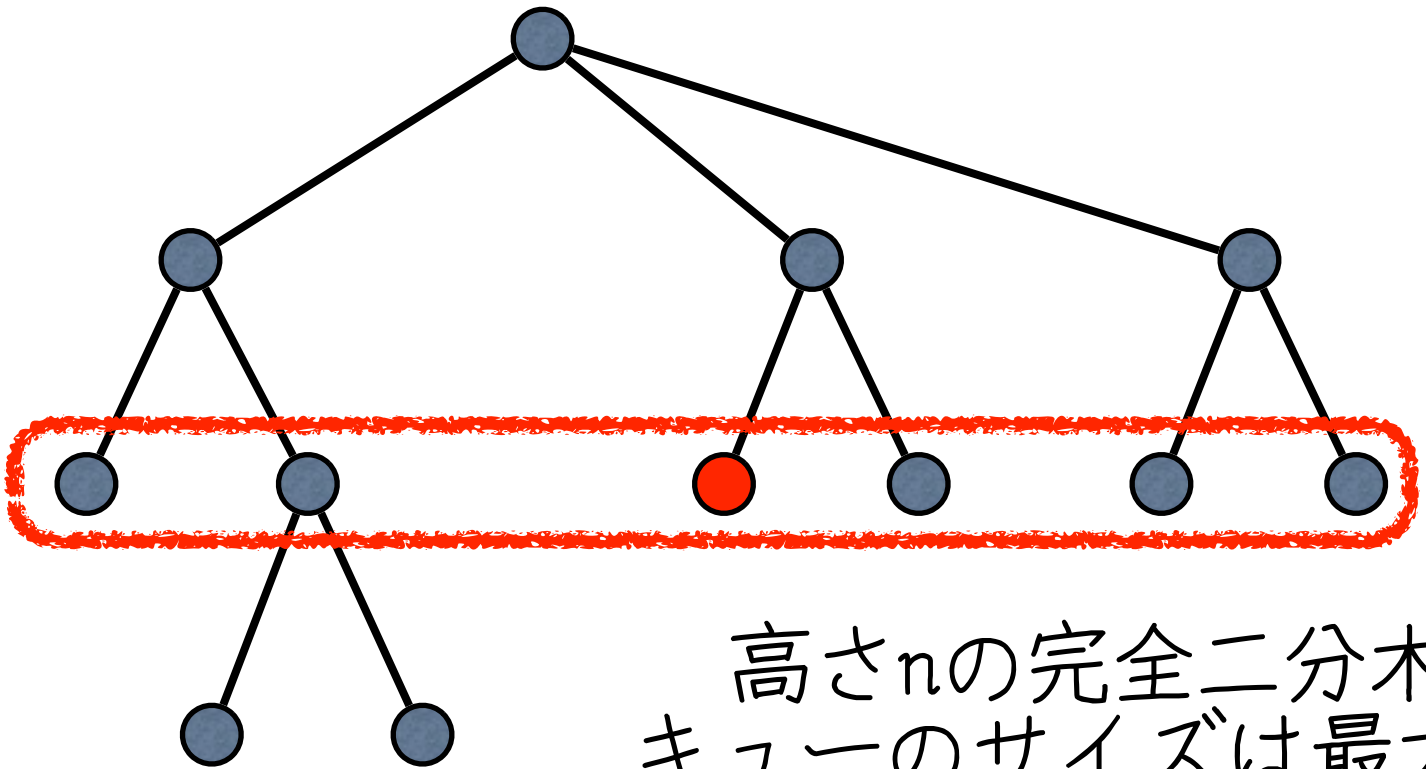
```
bfs :: SearchT a -> Maybe a
bfs t = case [ x | SUnit x <- queue ] of
          []  -> Nothing
          a:_ -> Just a
  where
    queue = t:runBFS 1 queue
    runBFS n ts
      | n == 0 = []
      | n > 0  =
        case ts of
          SOr t1 t2:ts' -> t1:t2:runBFS (n+1) ts'
          _:ts'         -> runBFS (n-1) ts'
```

利点・欠点

- 利点
  - 実装が比較的容易
    - dfsのスタックの代わりにキュー
  - みつかった解は最短手順で到達可
  - 解があれば必ず発見
    - 無限の木があっても動く
- 欠点
  - 空間効率が悪い

幅優先探索の空間効率

- キューのサイズは幅程度



高さnの完全二分木だと
キューのサイズは最大で2^nに

空間効率は大事

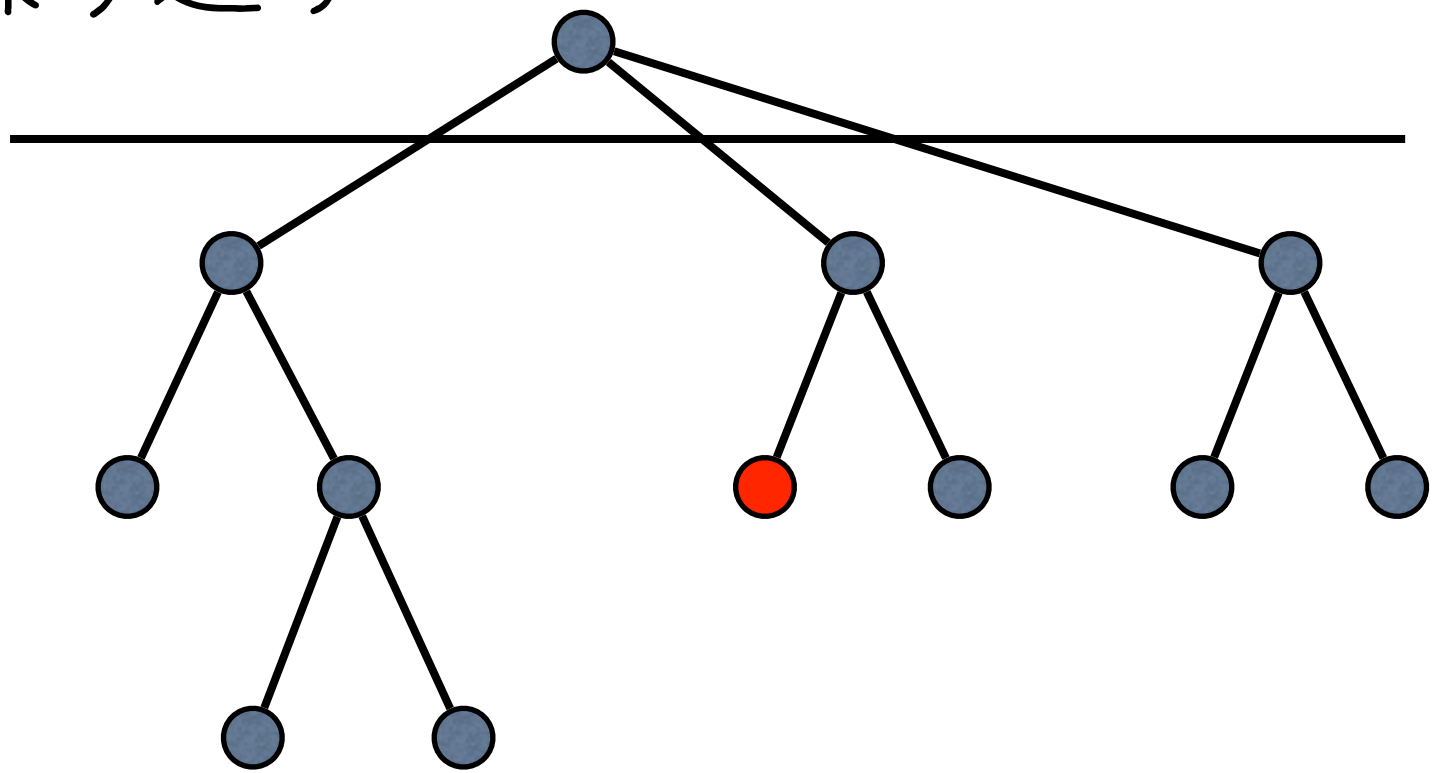
- かかる時間が
  1時間，2時間，3時間，…
  - 基本「同じマシン」で実行でき，
    待つ時間が増えるだけ
- 使用するメモリが
  1GB，2GB，3GB，…
  - 途中で「今つかっているマシン」では
    （現実的に）終わらなくなる
  - メモリのサイズを増やすのは
    待つ時間を増やすのより難しい

反復深化法

- 利点
  - 幅優先探索のように
    - 解があれば必ず発見
      - 無限の木があっても
    - みつかった解は最短手順で到達可
  - 深さ優先探索のように
    - 空間計算量は深さ分程度

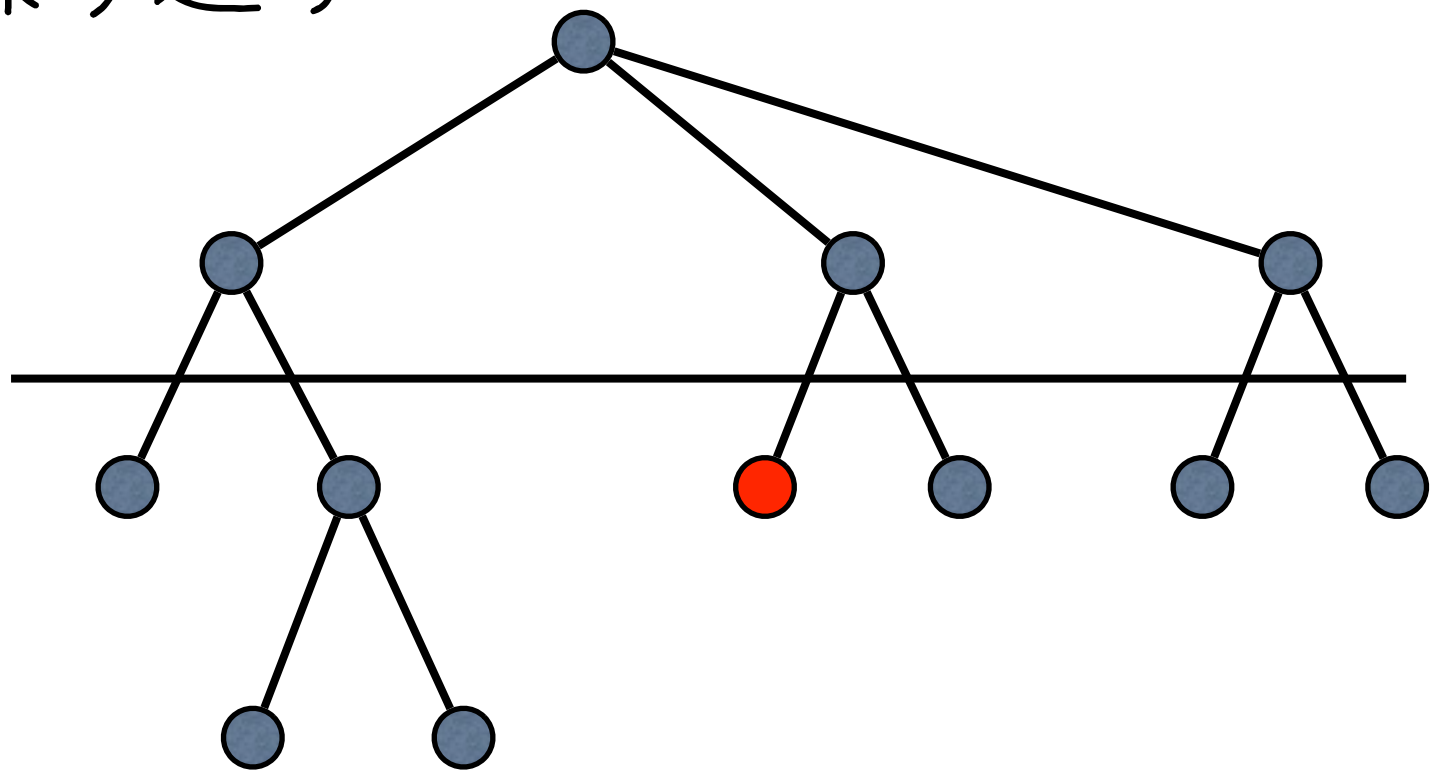
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



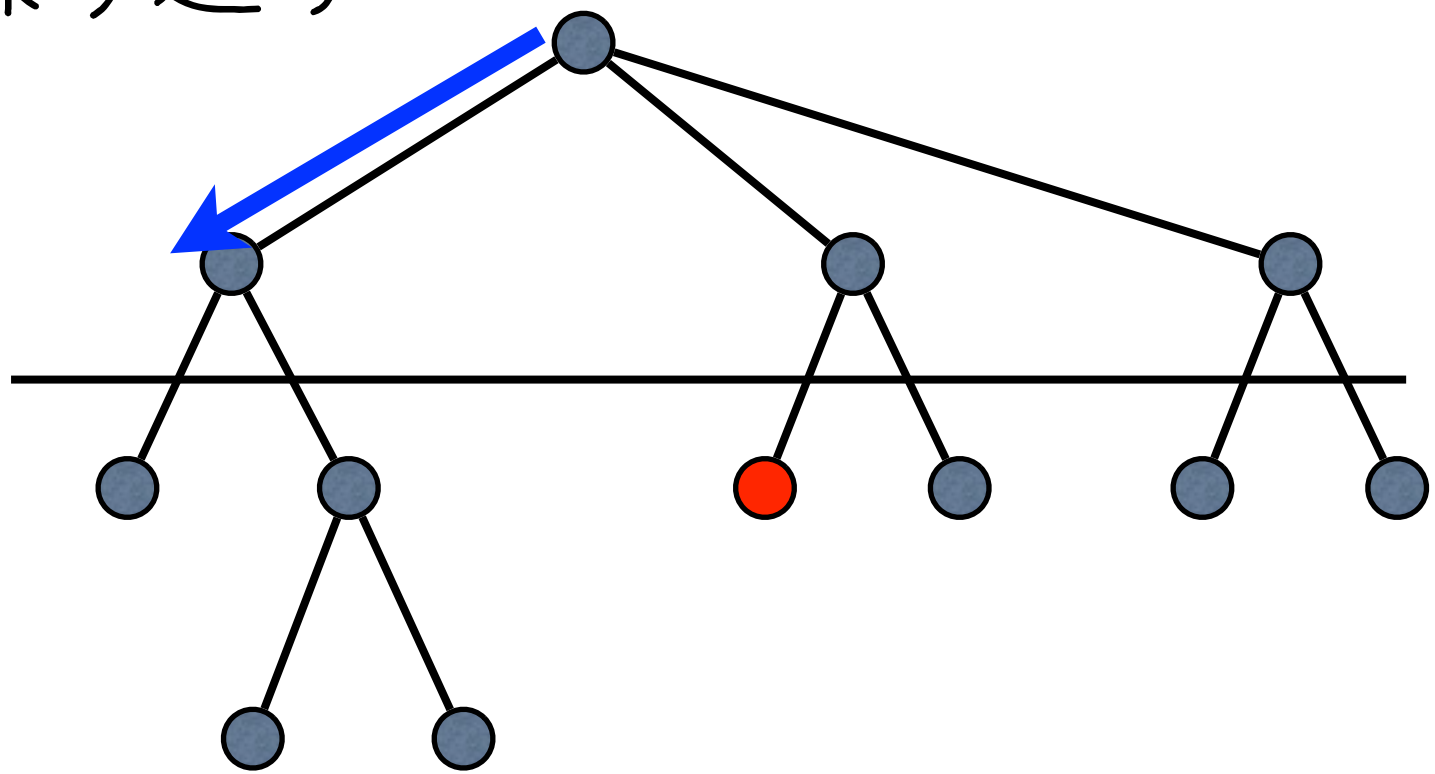
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



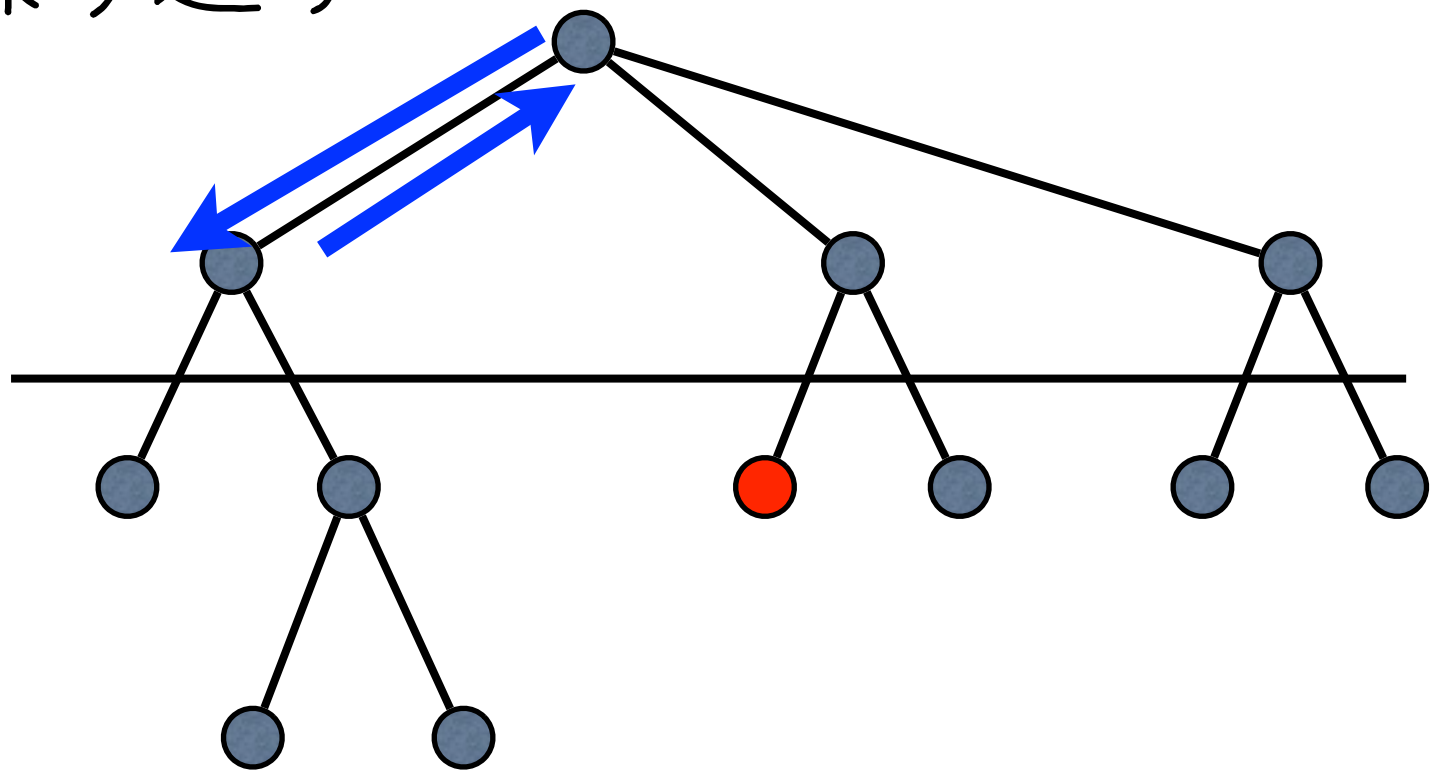
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



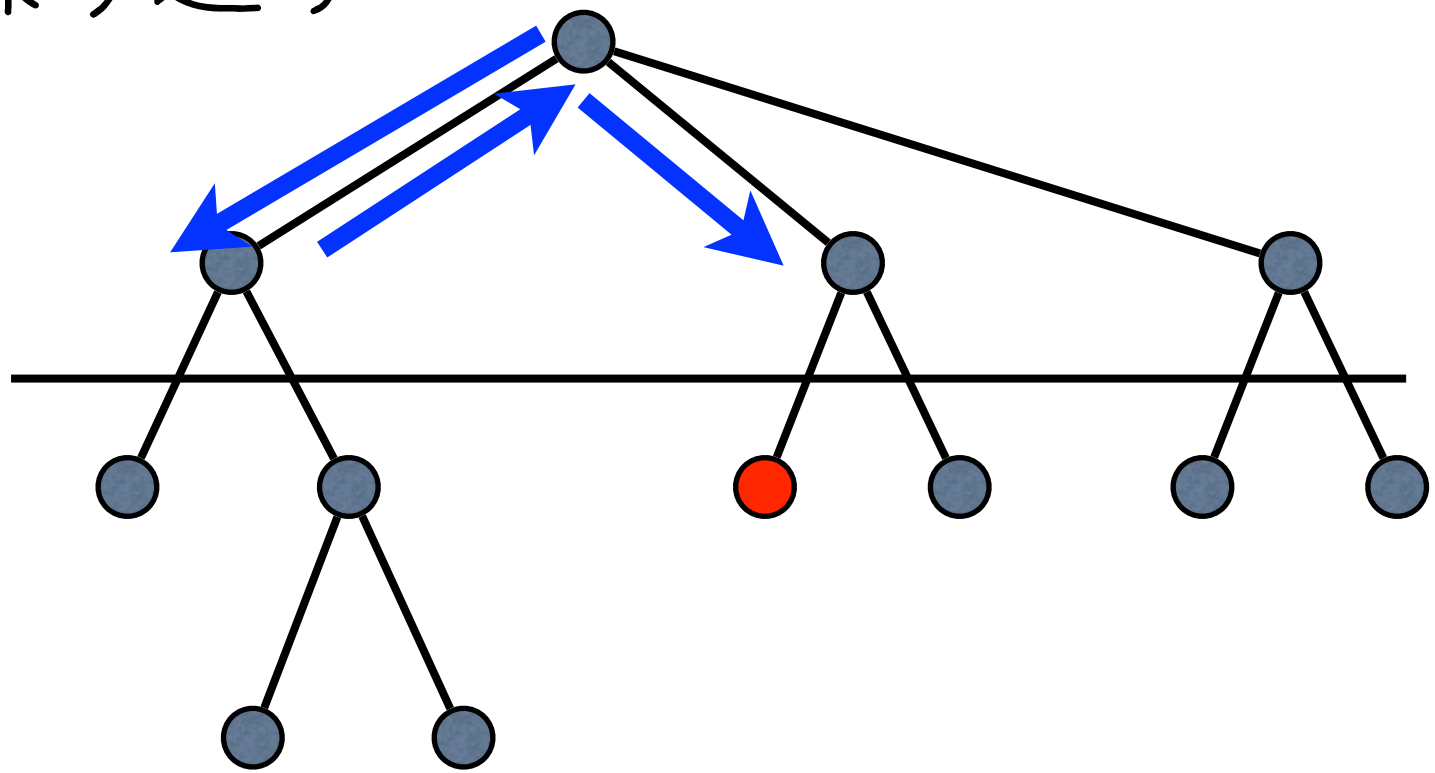
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



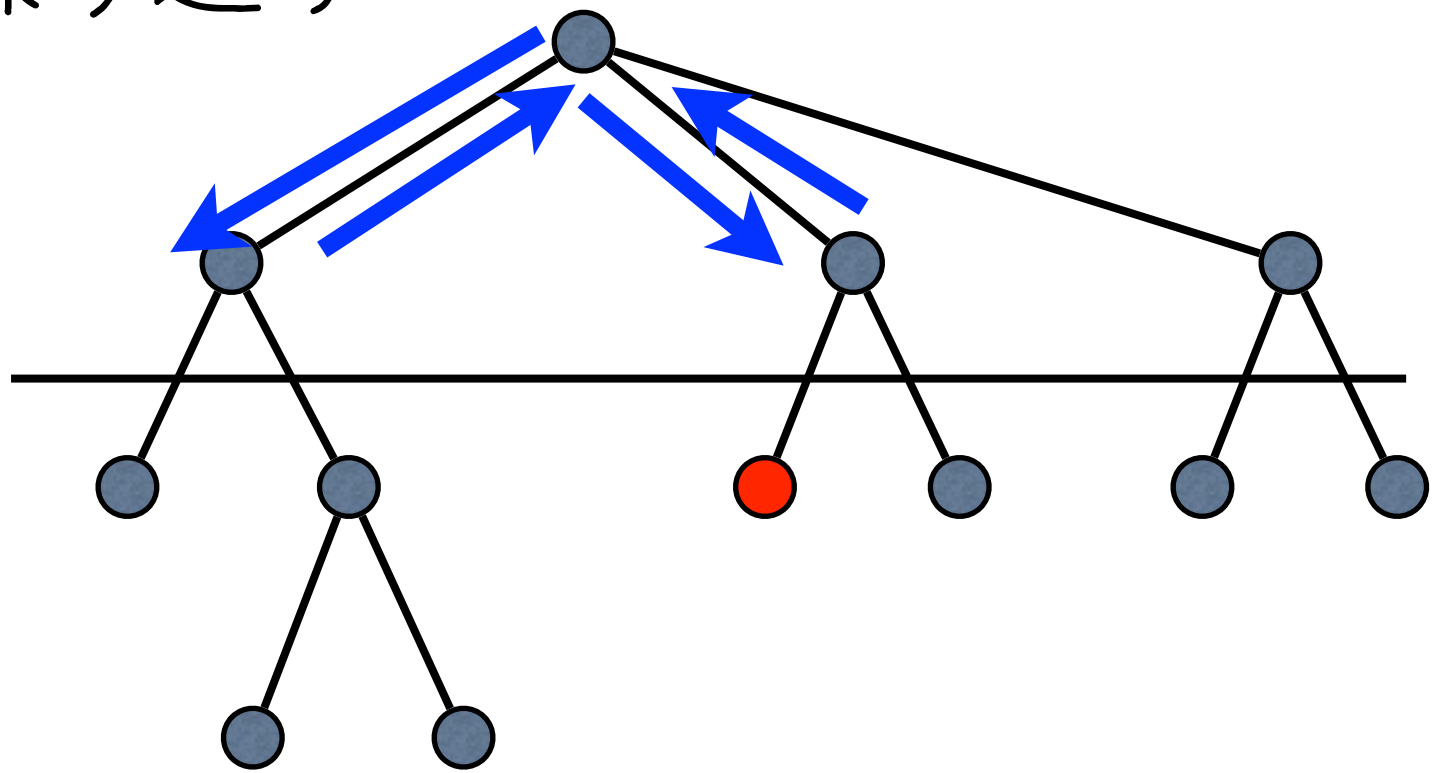
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



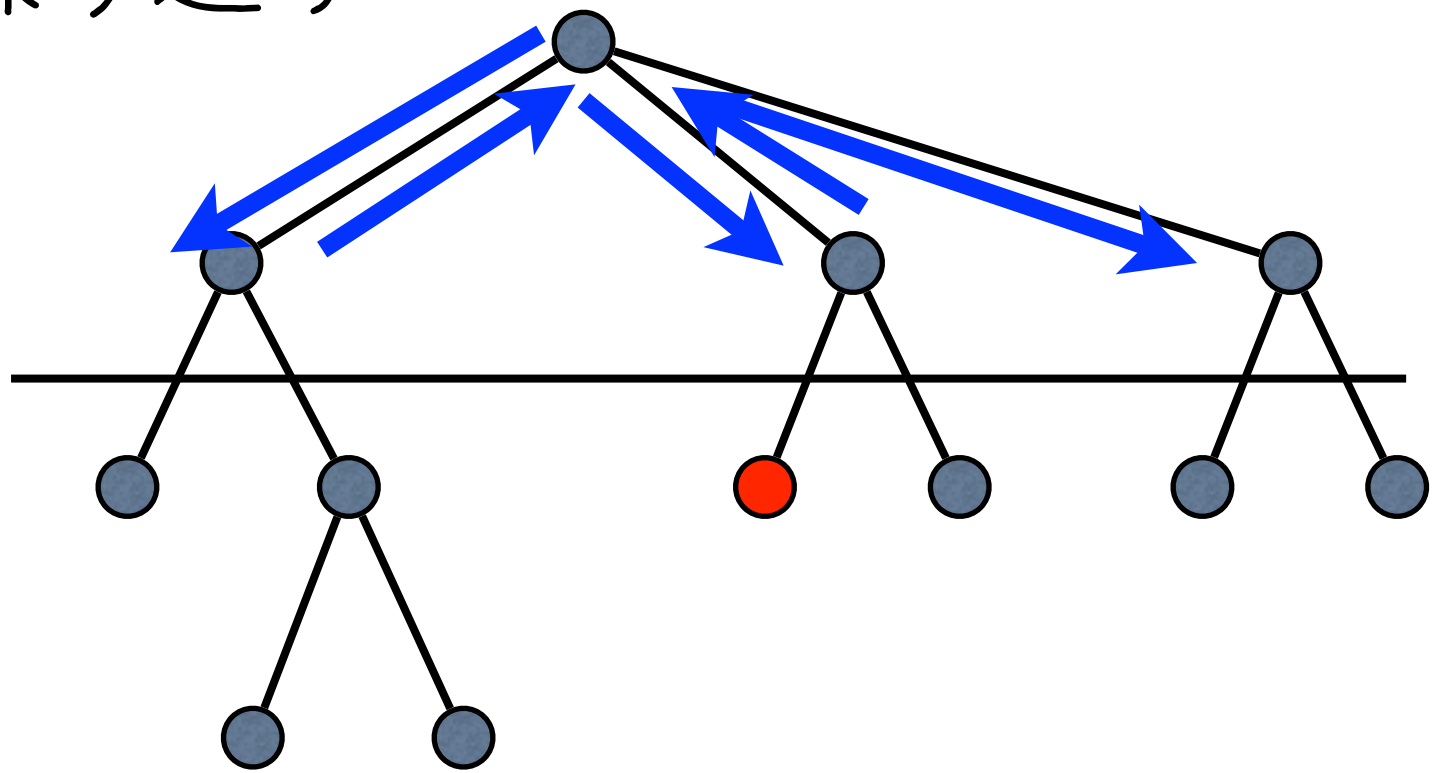
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



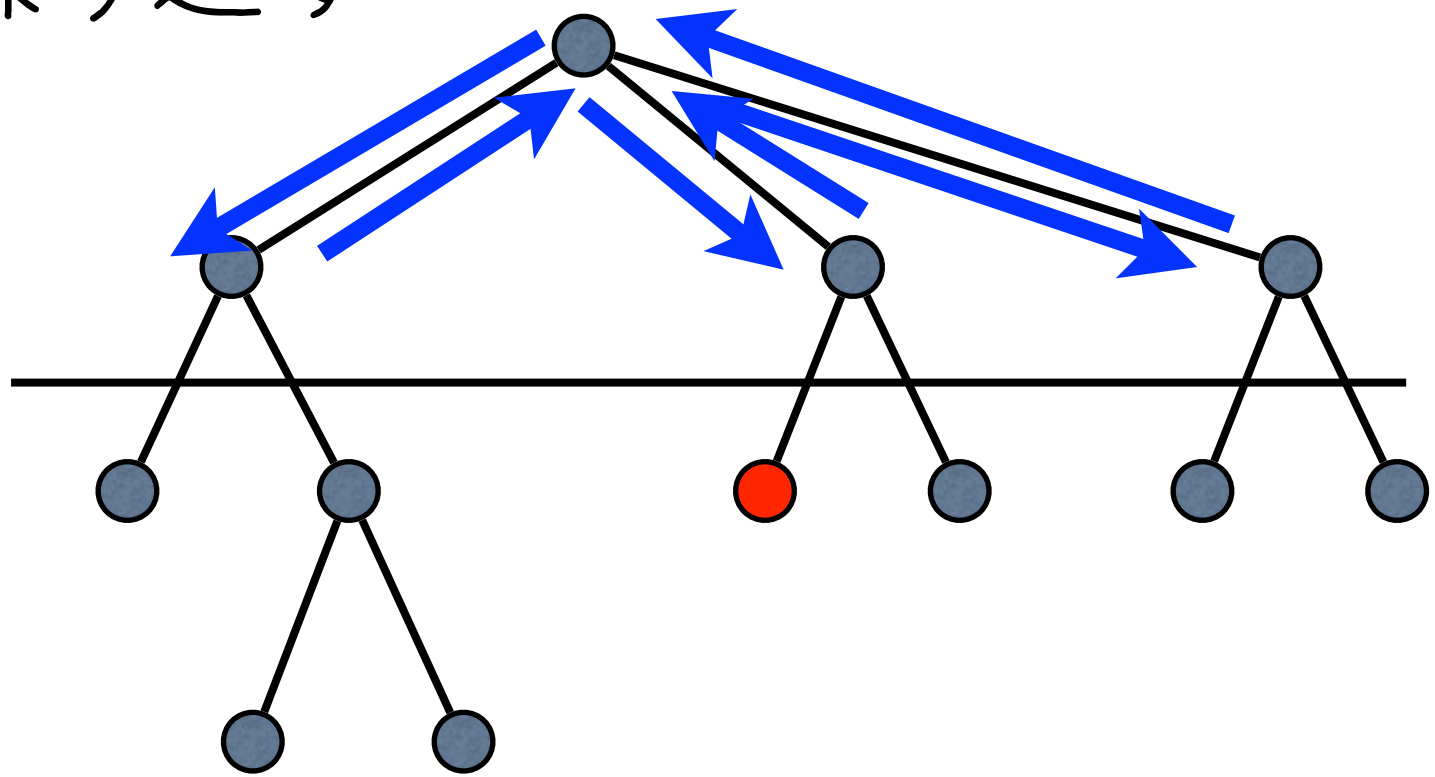
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



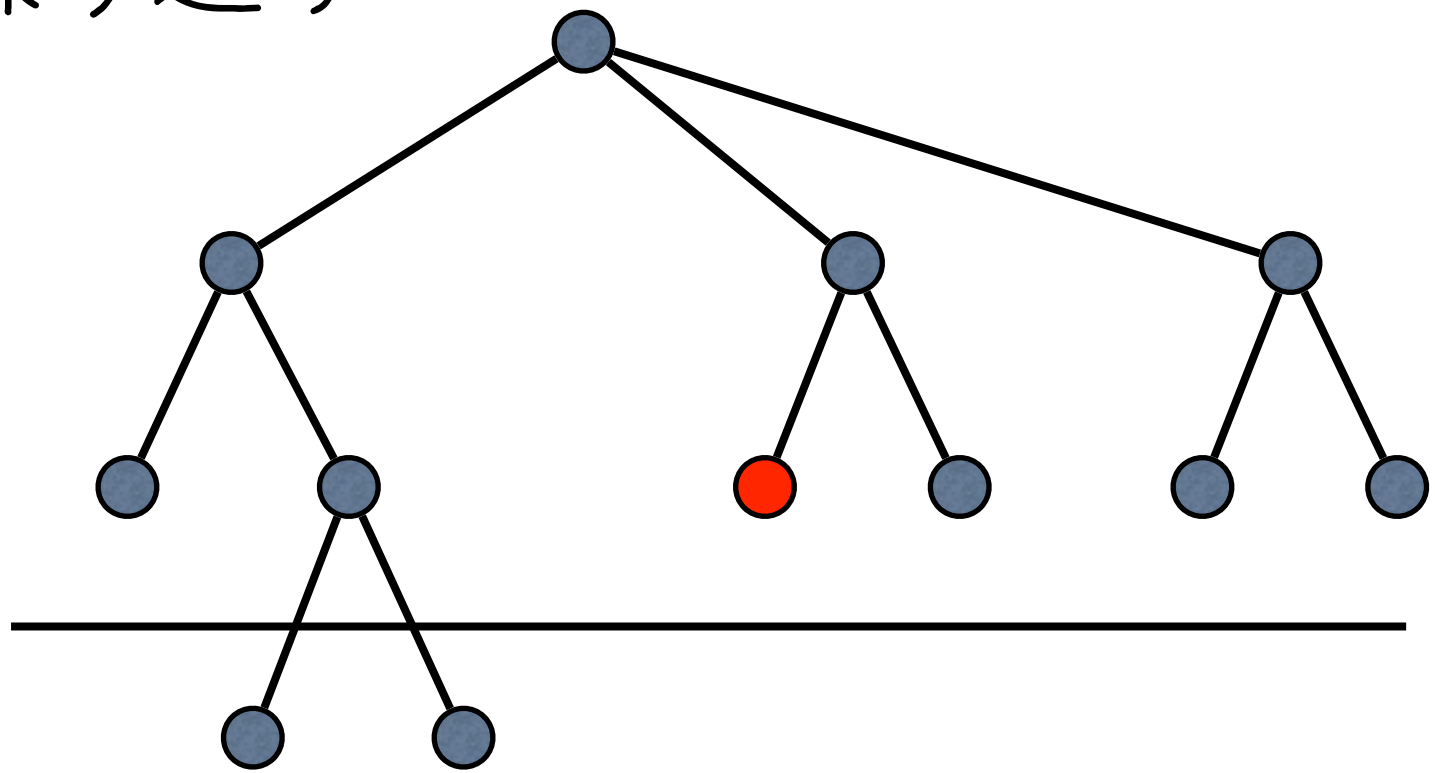
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



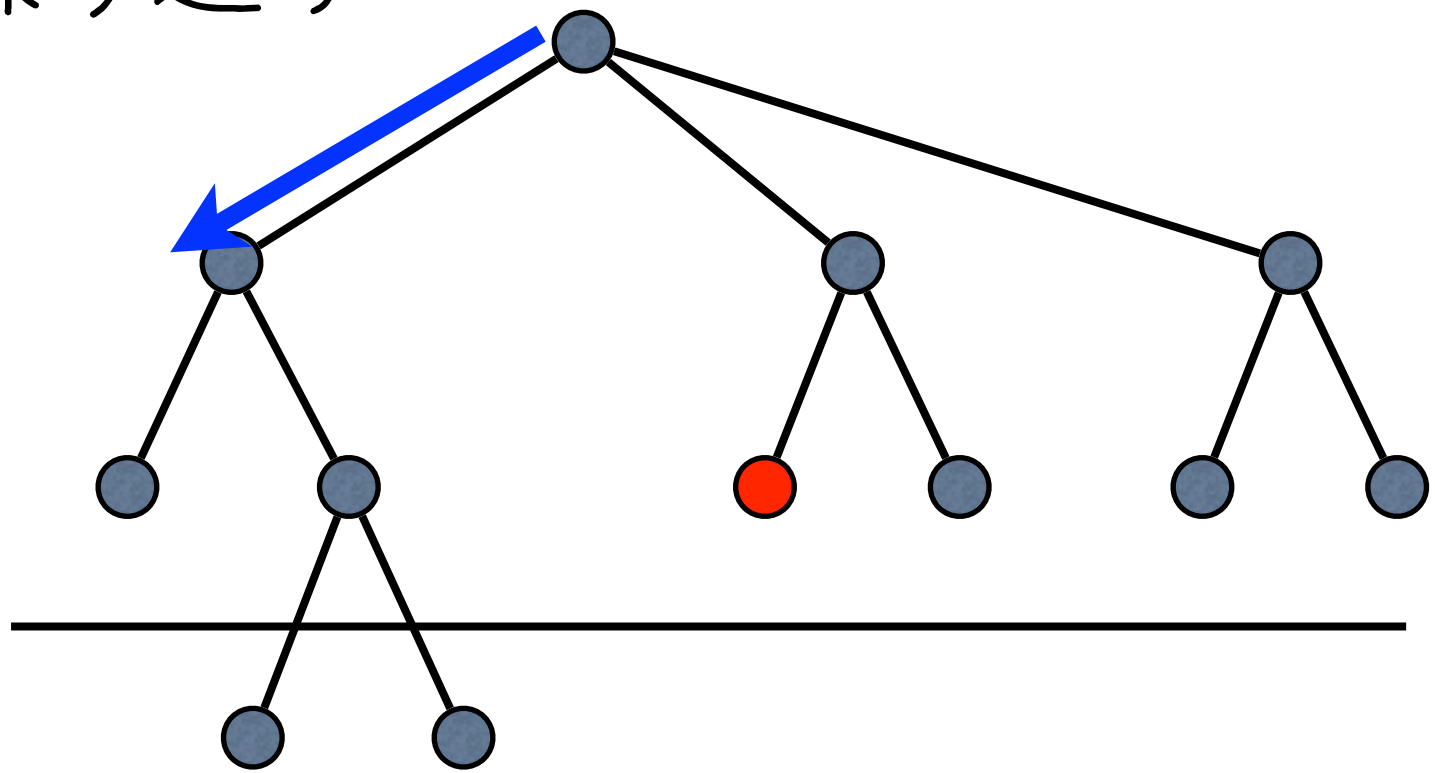
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



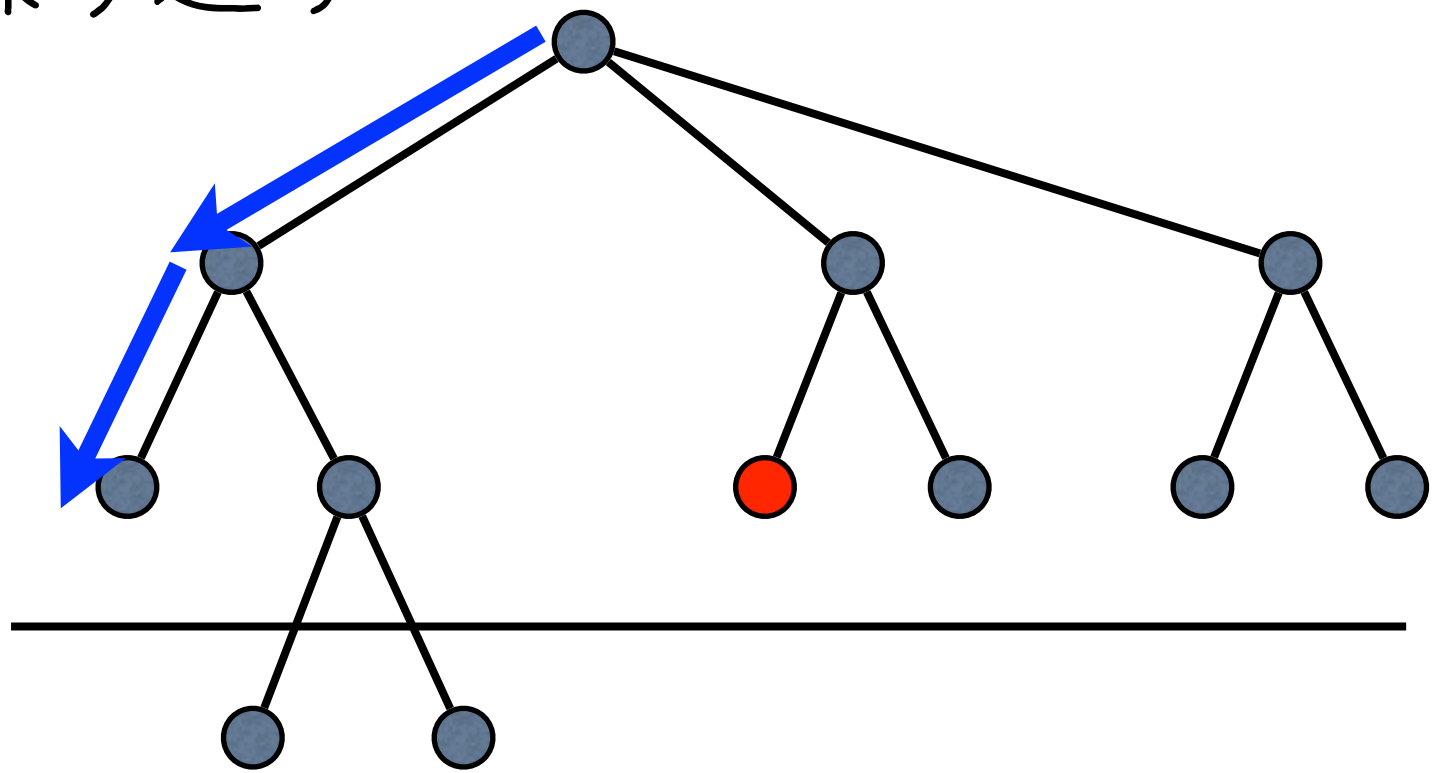
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



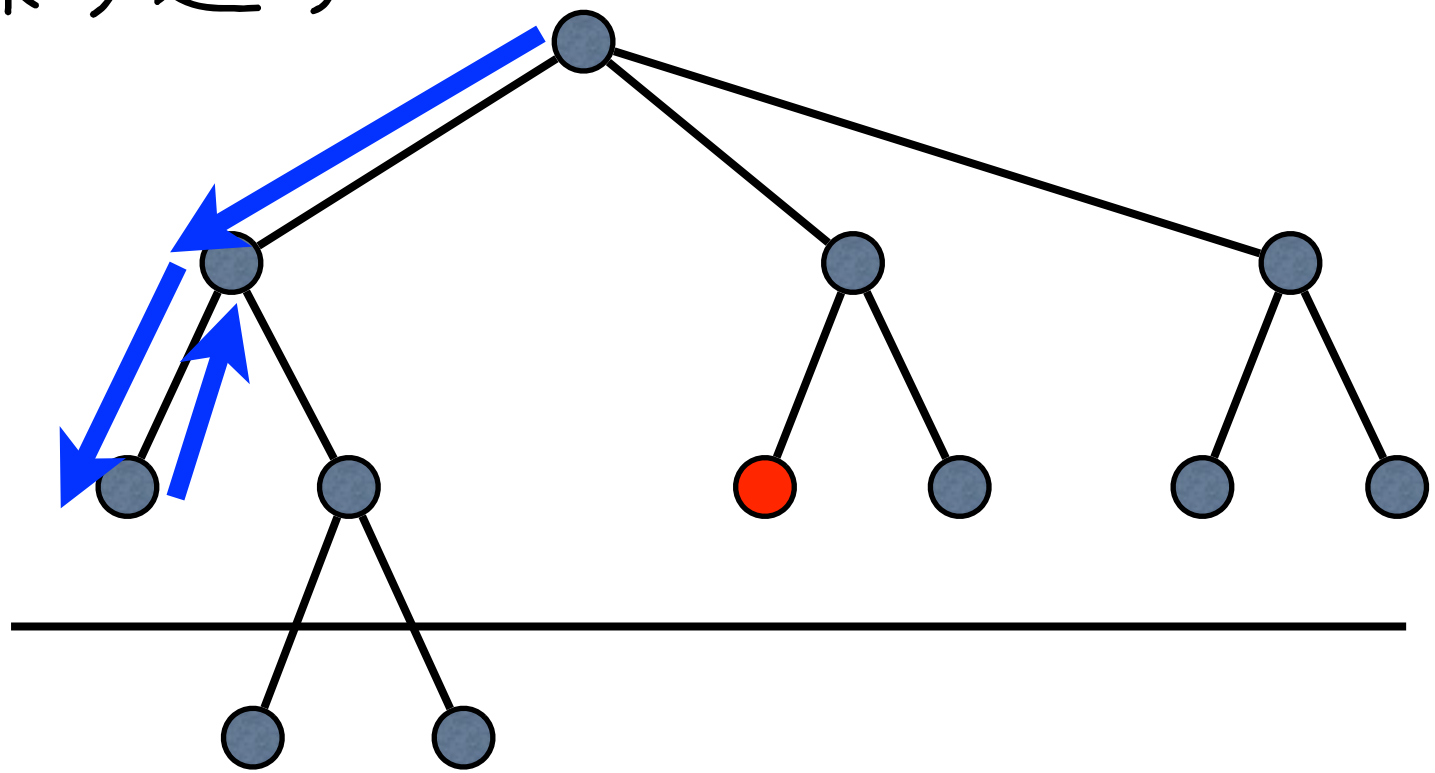
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



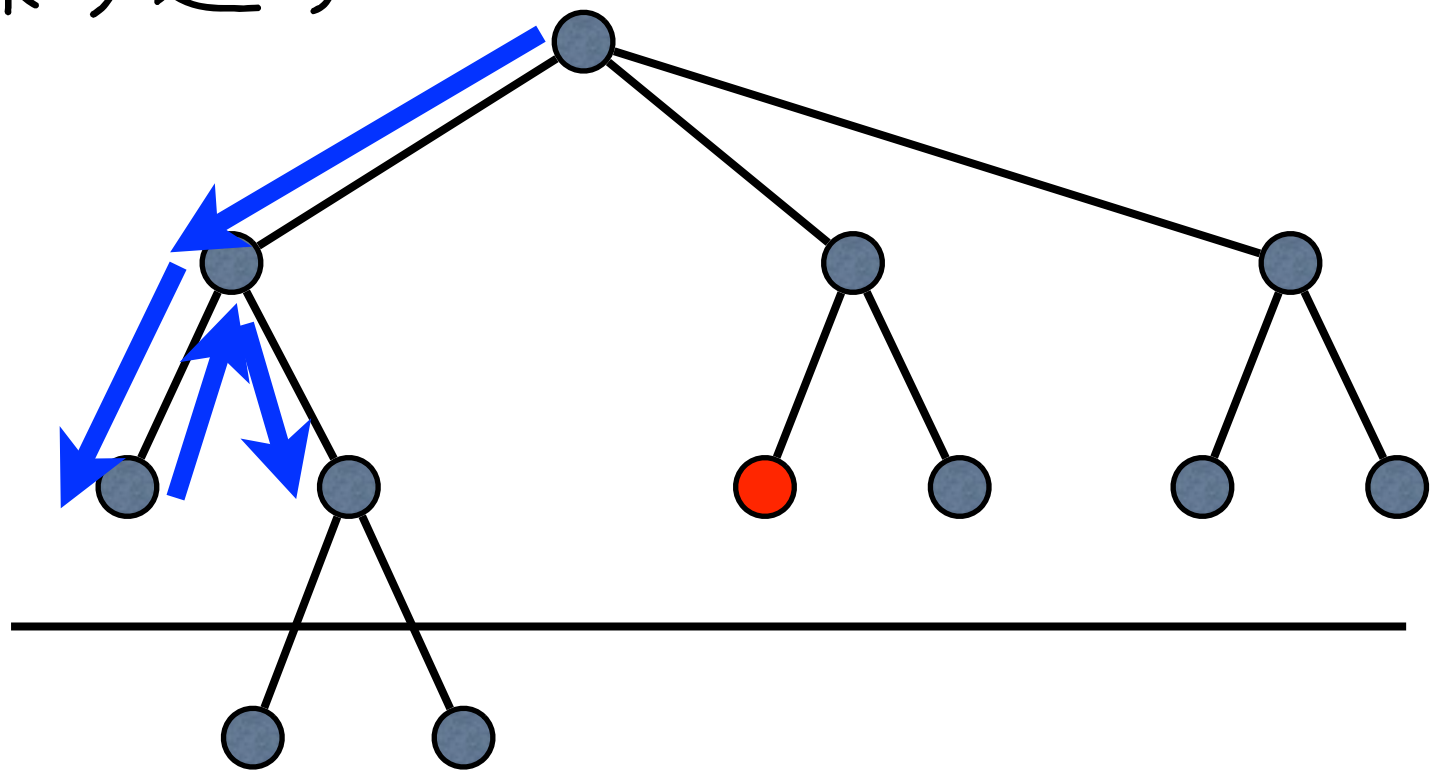
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



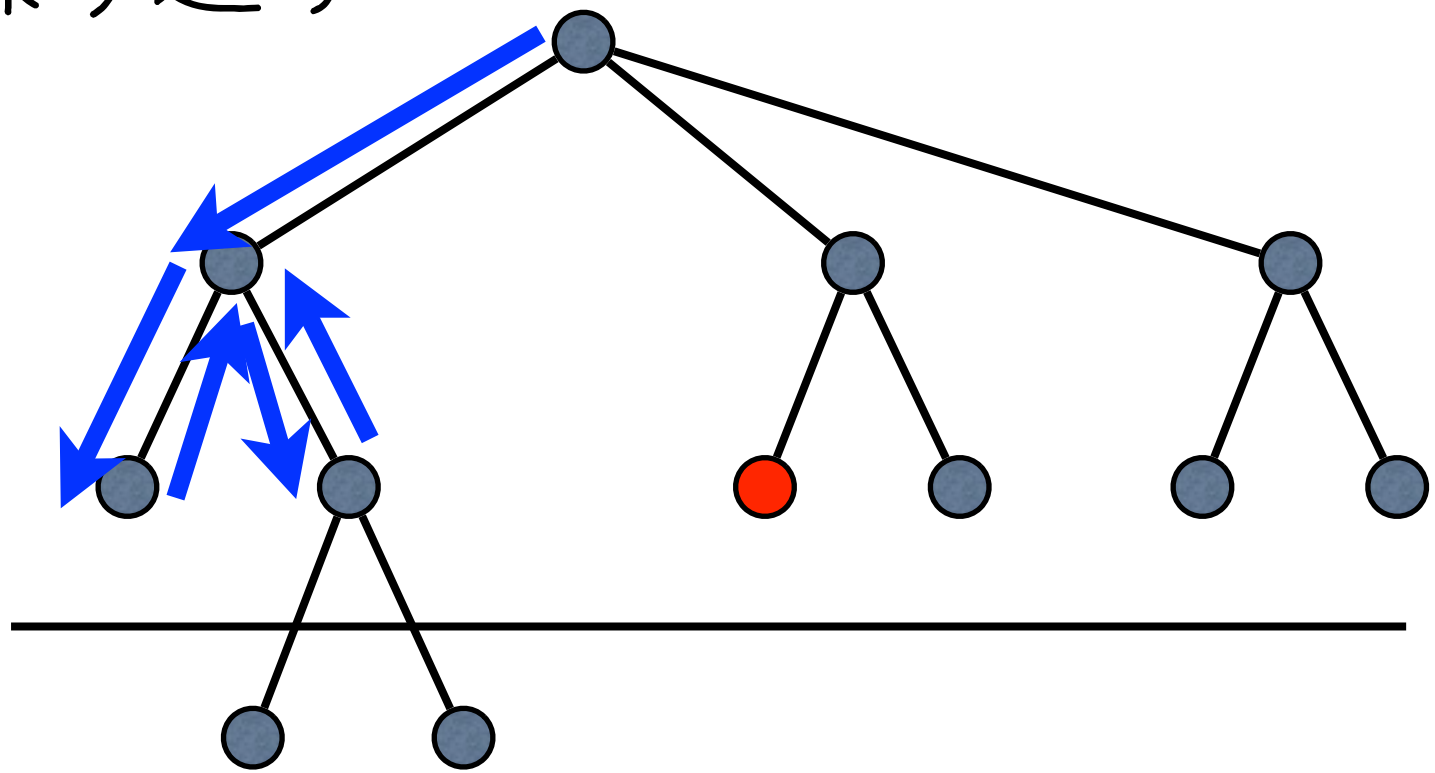
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



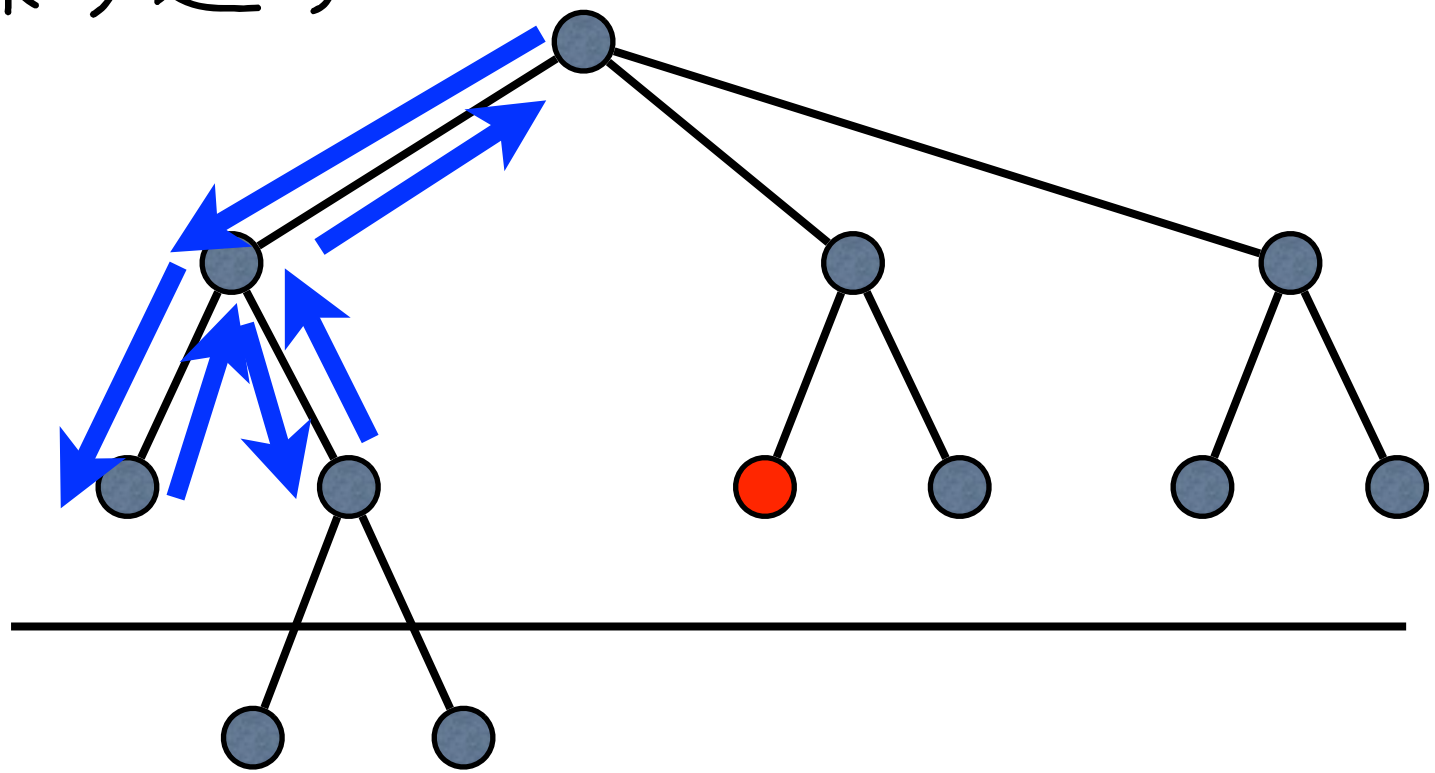
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



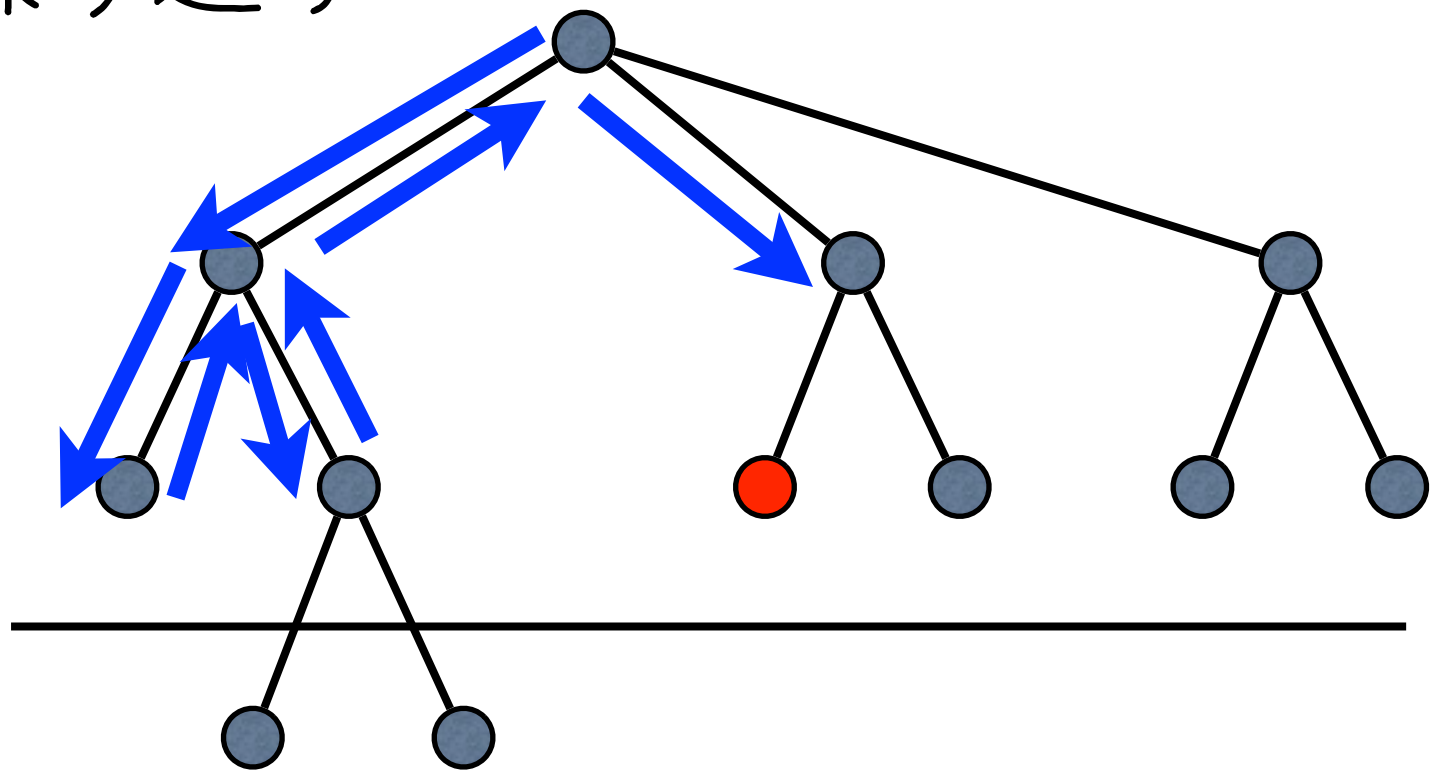
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



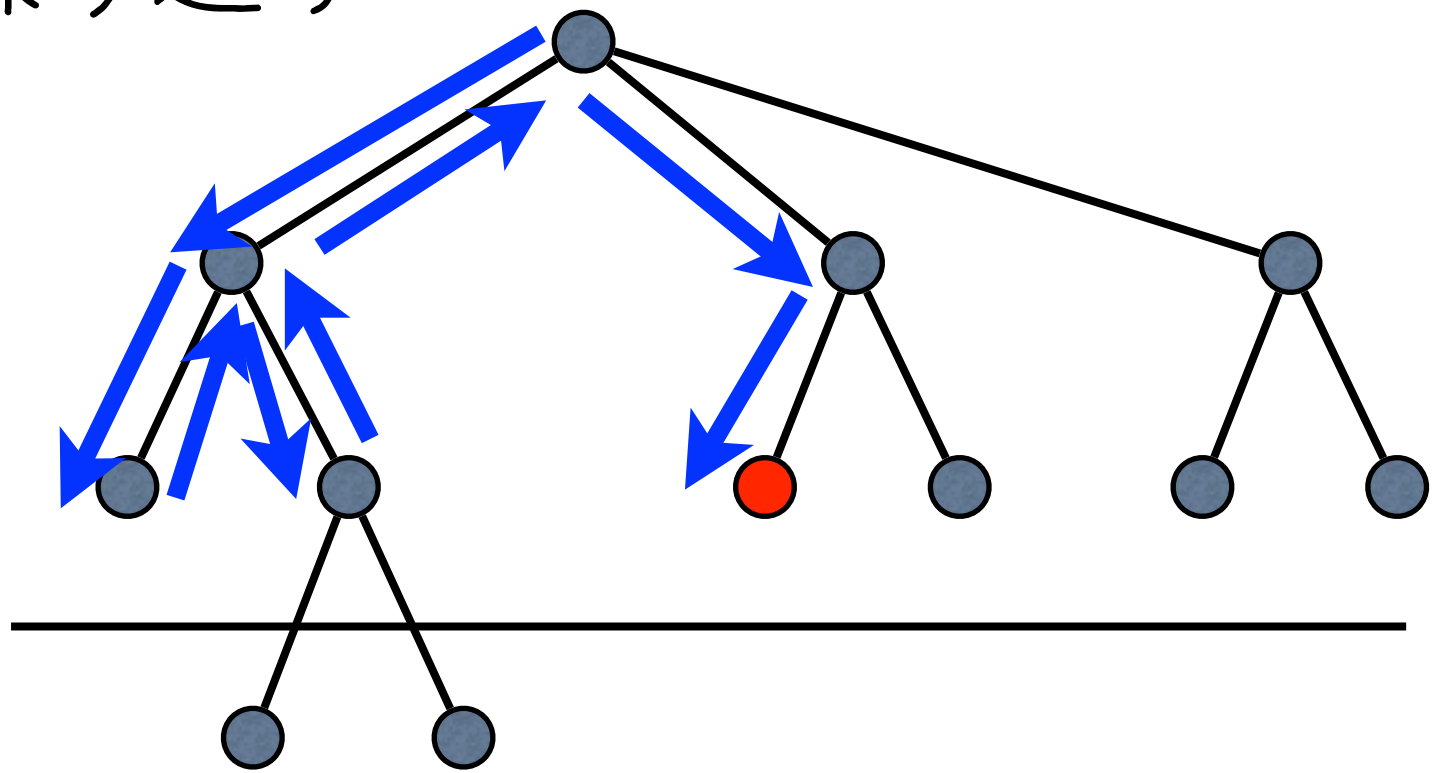
反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



反復深化法

- 探索する深さを決め，深さ優先探索を繰り返す



Haskellでの実装例

```
iddfs :: SearchT a -> Maybe a
iddfs t = foldr (\a _ -> Just a) Nothing
            $ concatMap (go [(t,0)]) [1..]
  where
    go [] limit = []
    go ((SNone,_):x) limit = go x limit
    go ((SUnit a,_):x) limit = [a]
    go ((SOr l r,n):x) limit =
       if n < limit then
         go ((l,n+1):(r,n+1):x) limit
       else
         go x limit
```

遅くない？

- 何度もDFSを実行
  - 同じノードを何度も何度も辿る
  - が，辿ったノードを覚えておくにはやっぱり幅程度の空間コストがかかる

遅くない！

- 実は同じノードを複数回辿るオーバヘッドは木の高さ程度
  - 高さnの完全二分木について
    - 深さ優先探索のノードを辿る回数
      $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$
    - 反復深化法のノードを辿る回数
      $n + (n-1)2 + (n-2)4 + \cdots + 2^n$
      $\leqq n(1 + 2 + 4 + \cdots + 2^n)$
      $= n(2^{n+1} - 1)$

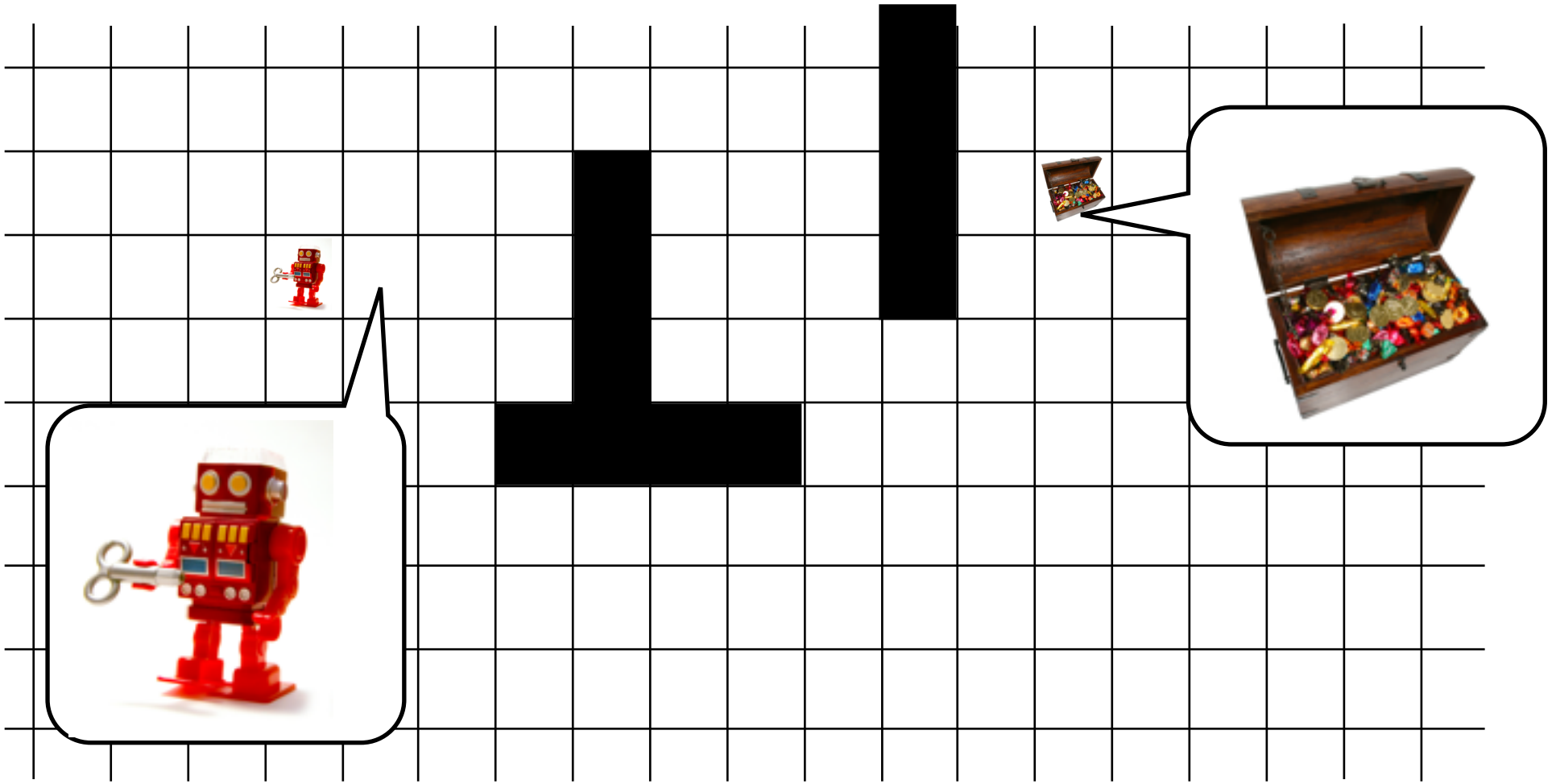
小まとめ

- 深さ優先探索
- 幅優先探索
- 反復深化法

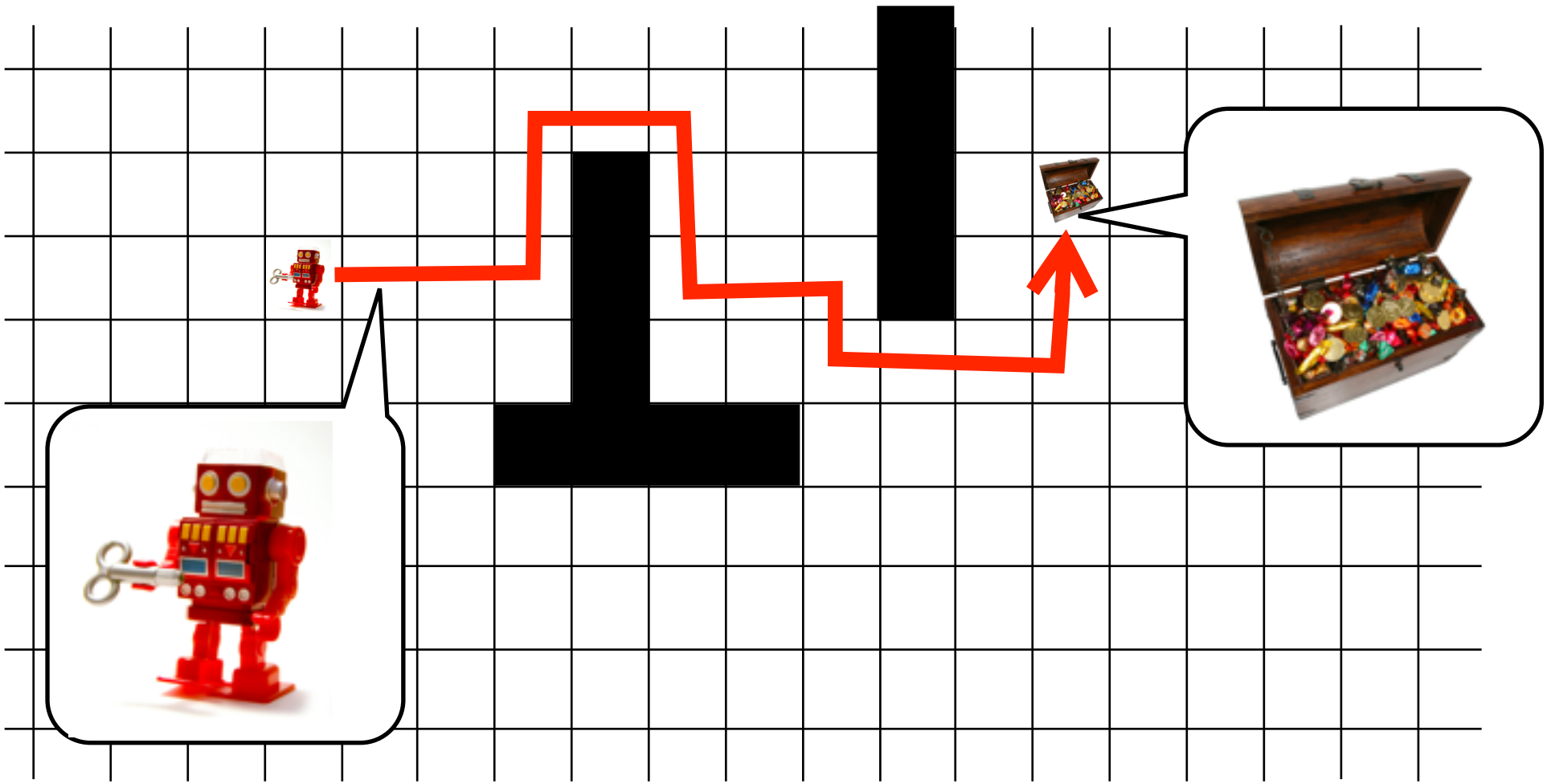
これまでの方法

- 問題と独立
- 枝に重みなし

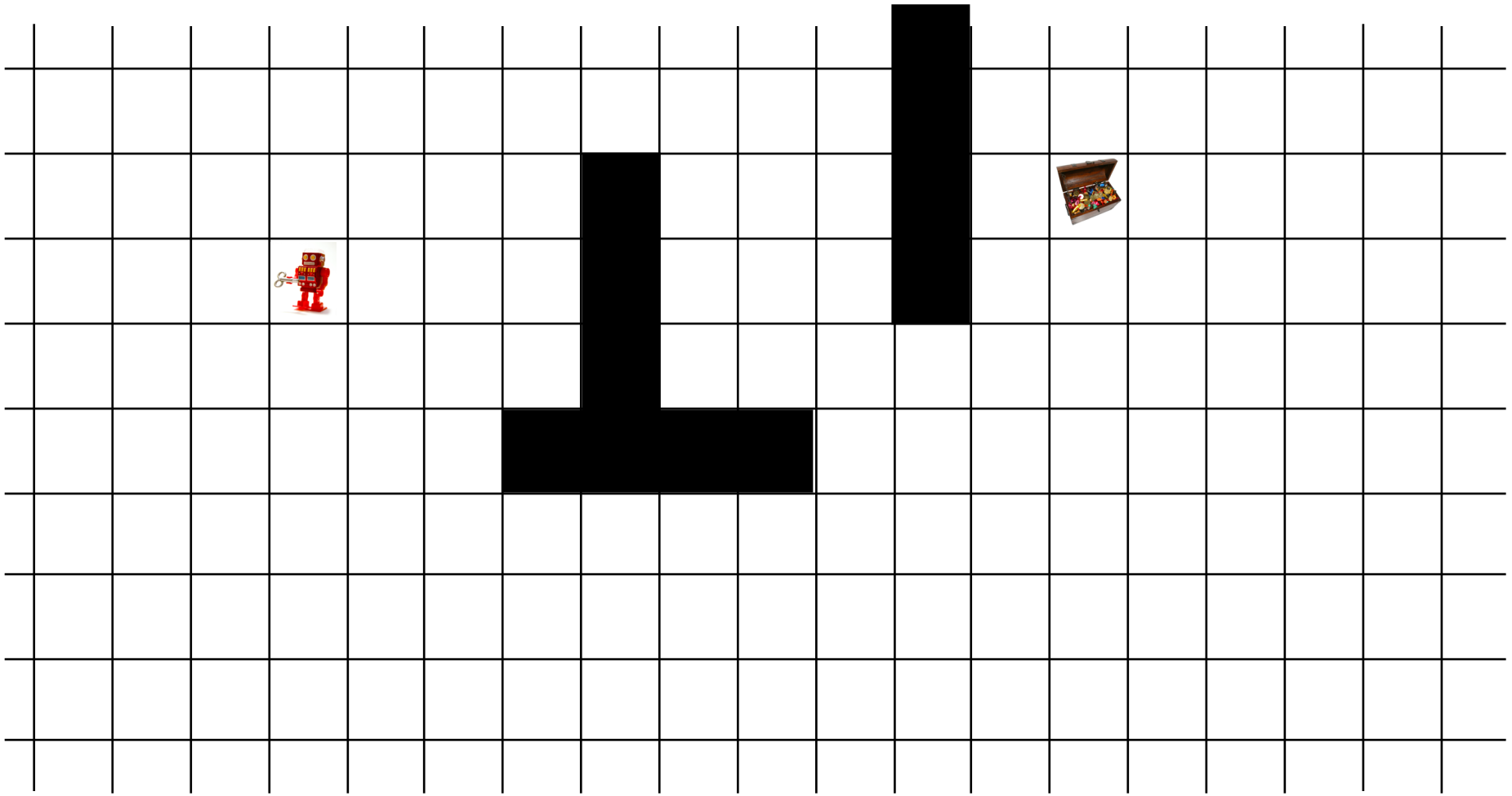
例：ロボットの移動



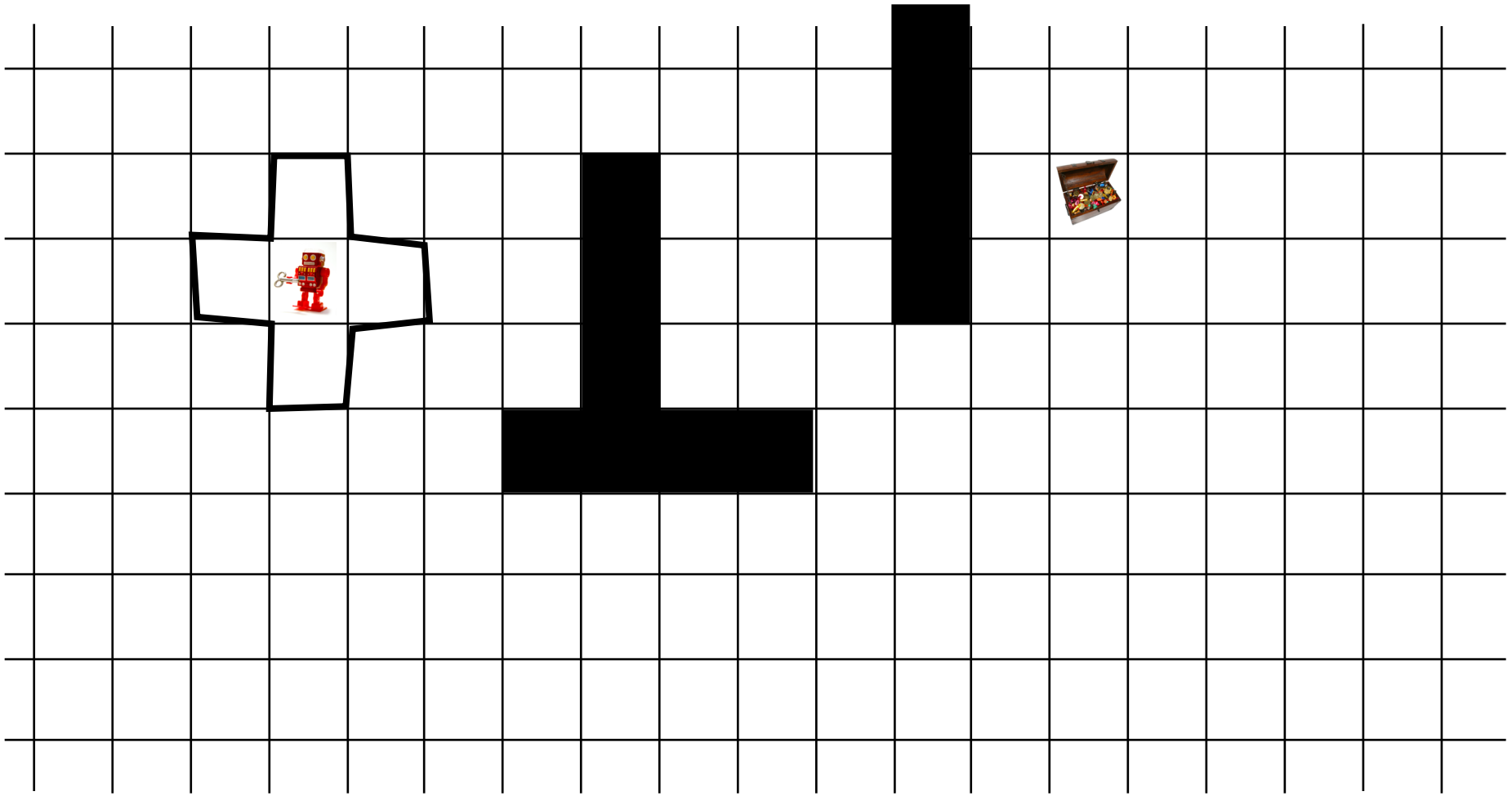
例：ロボットの移動



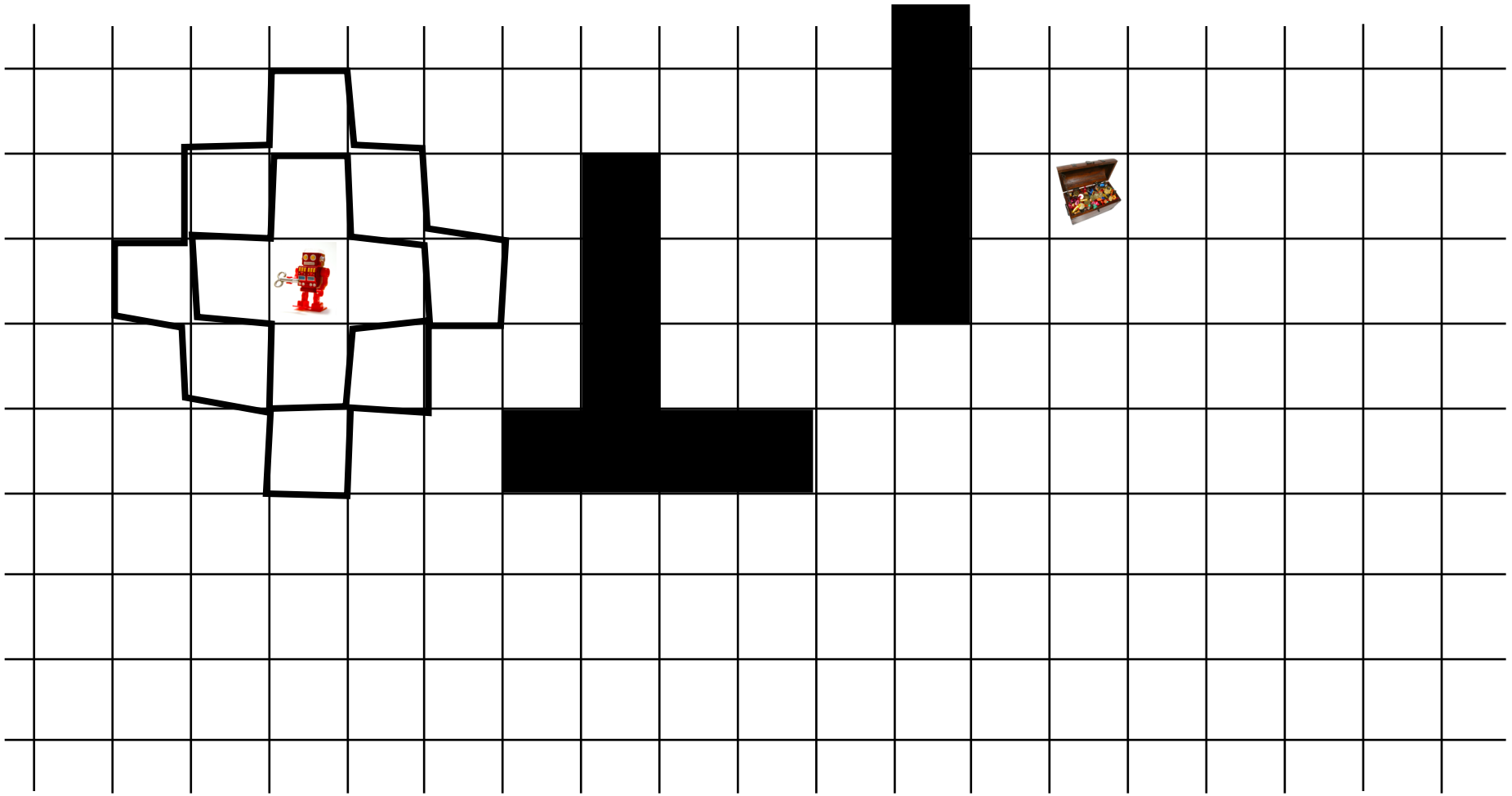
幅優先探索（Dijkstra法）



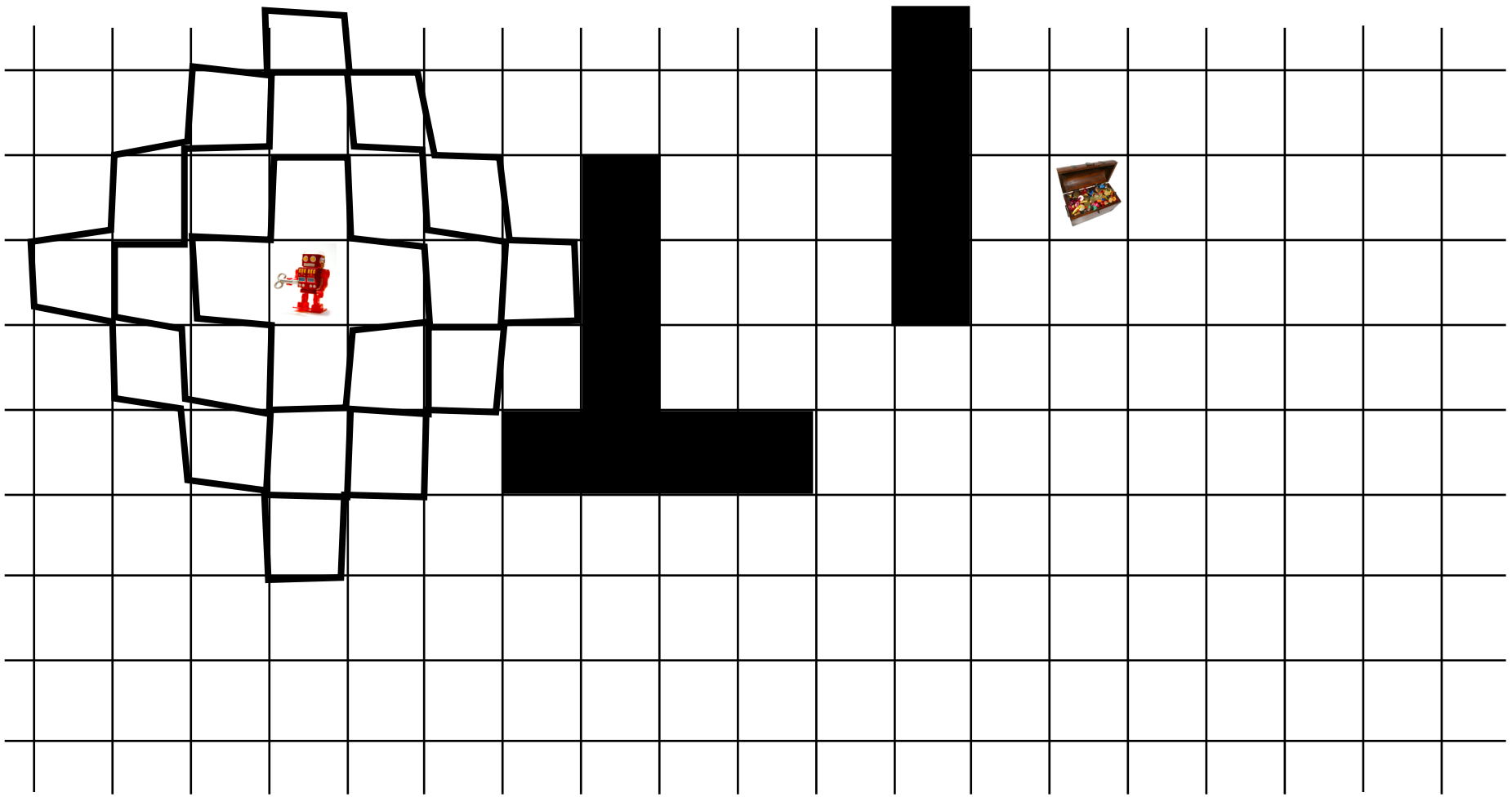
幅優先探索（Dijkstra法）



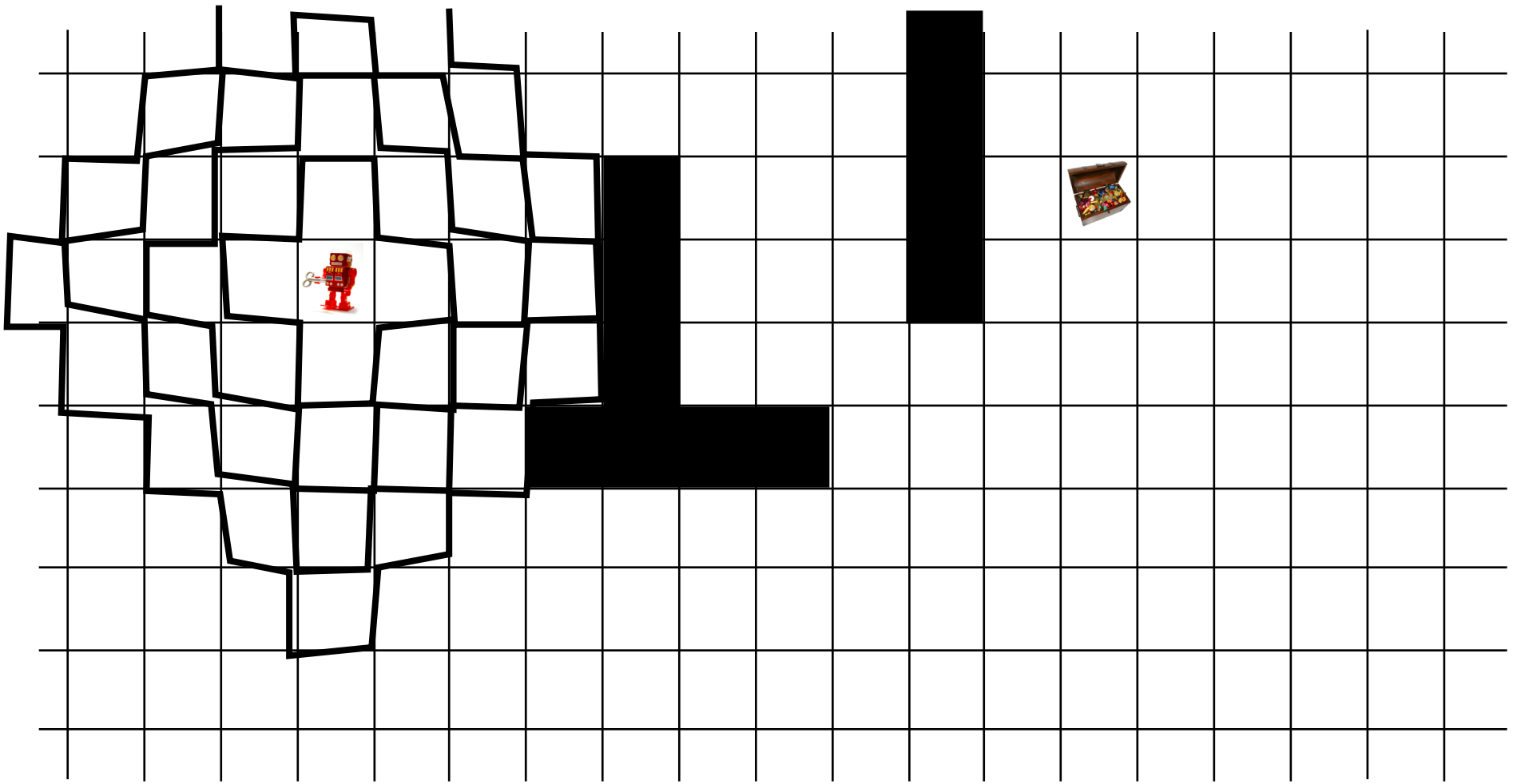
幅優先探索（Dijkstra法）



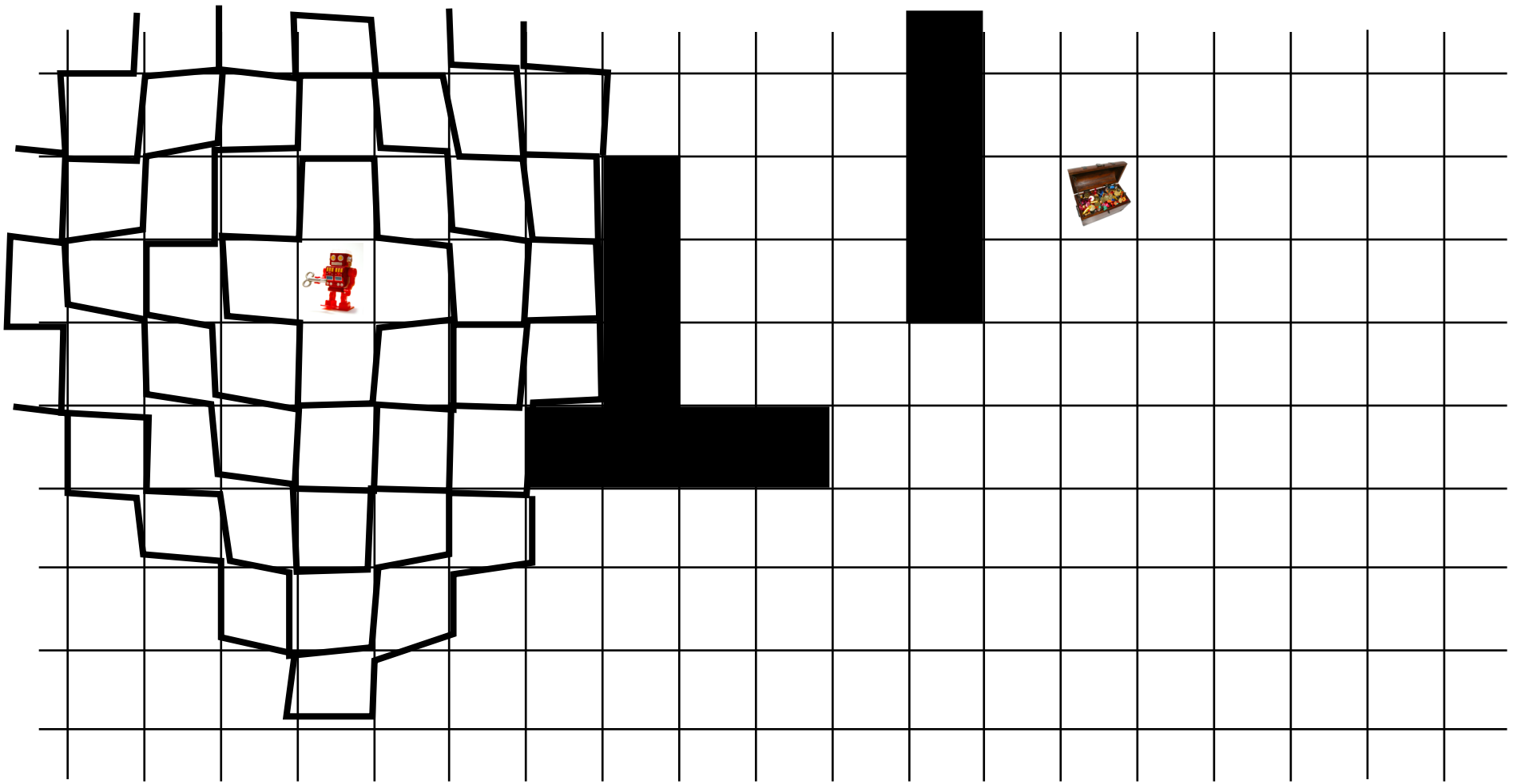
幅優先探索（Dijkstra法）



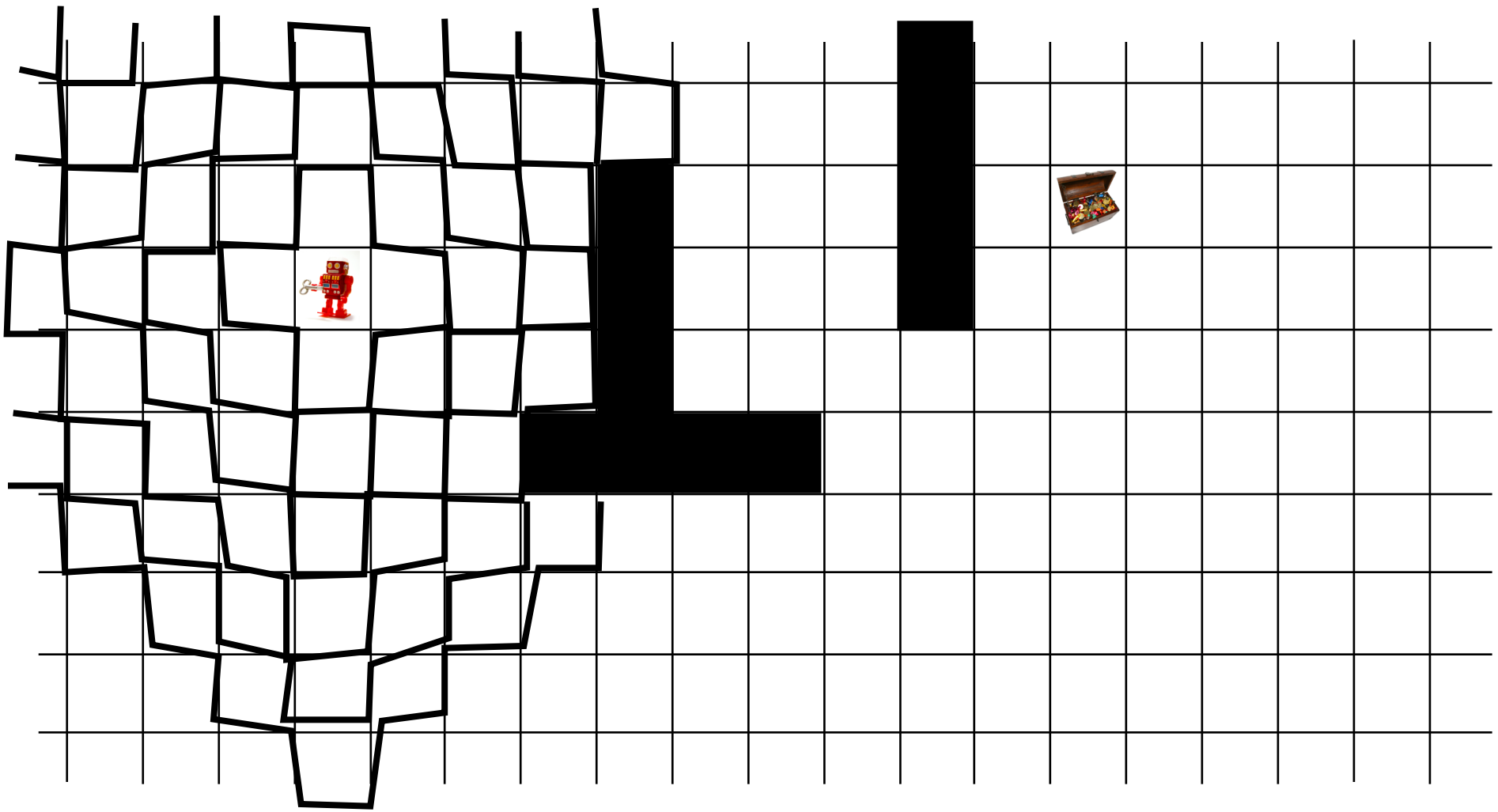
幅優先探索（Dijkstra法）



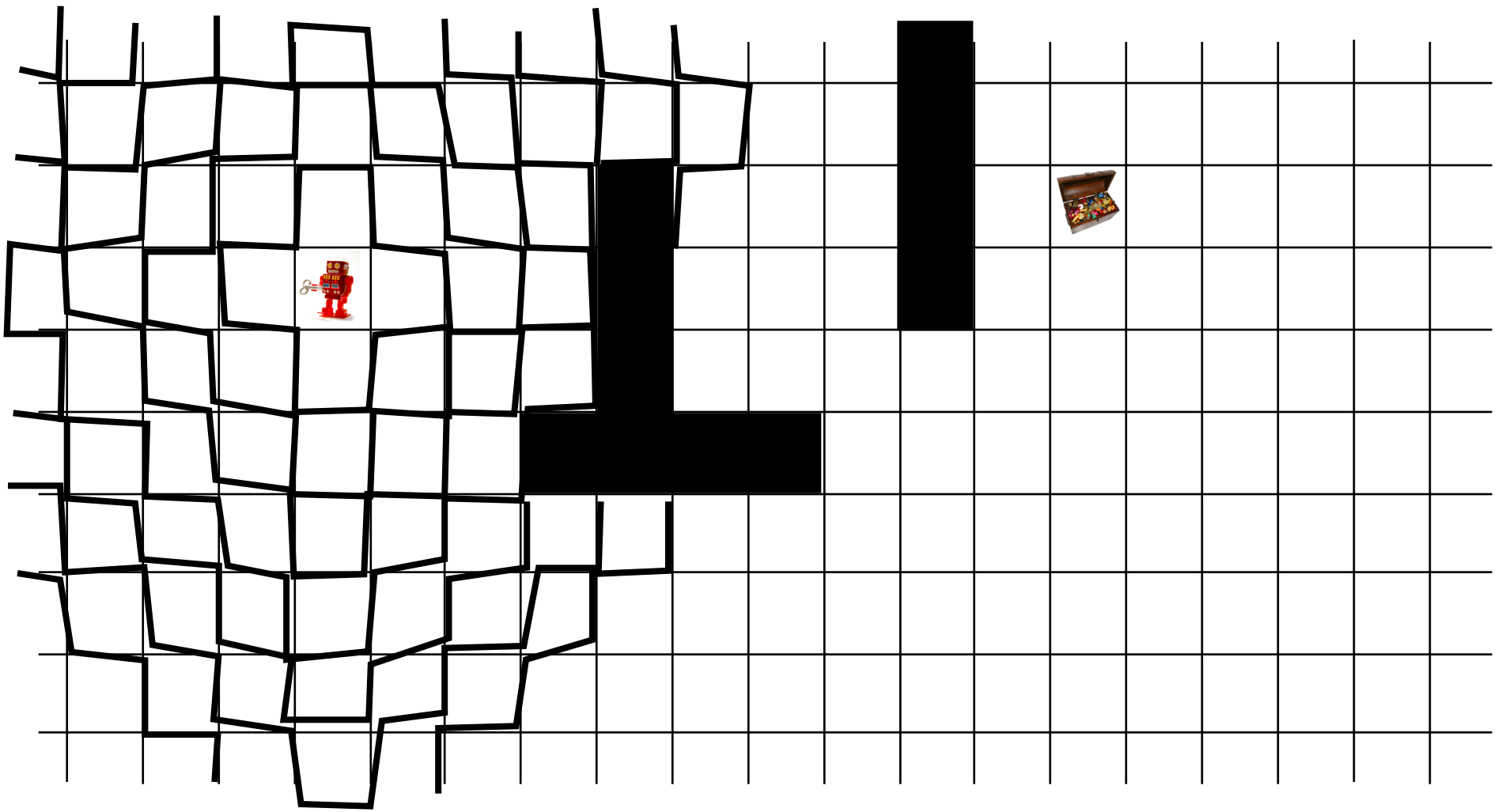
幅優先探索（Dijkstra法）



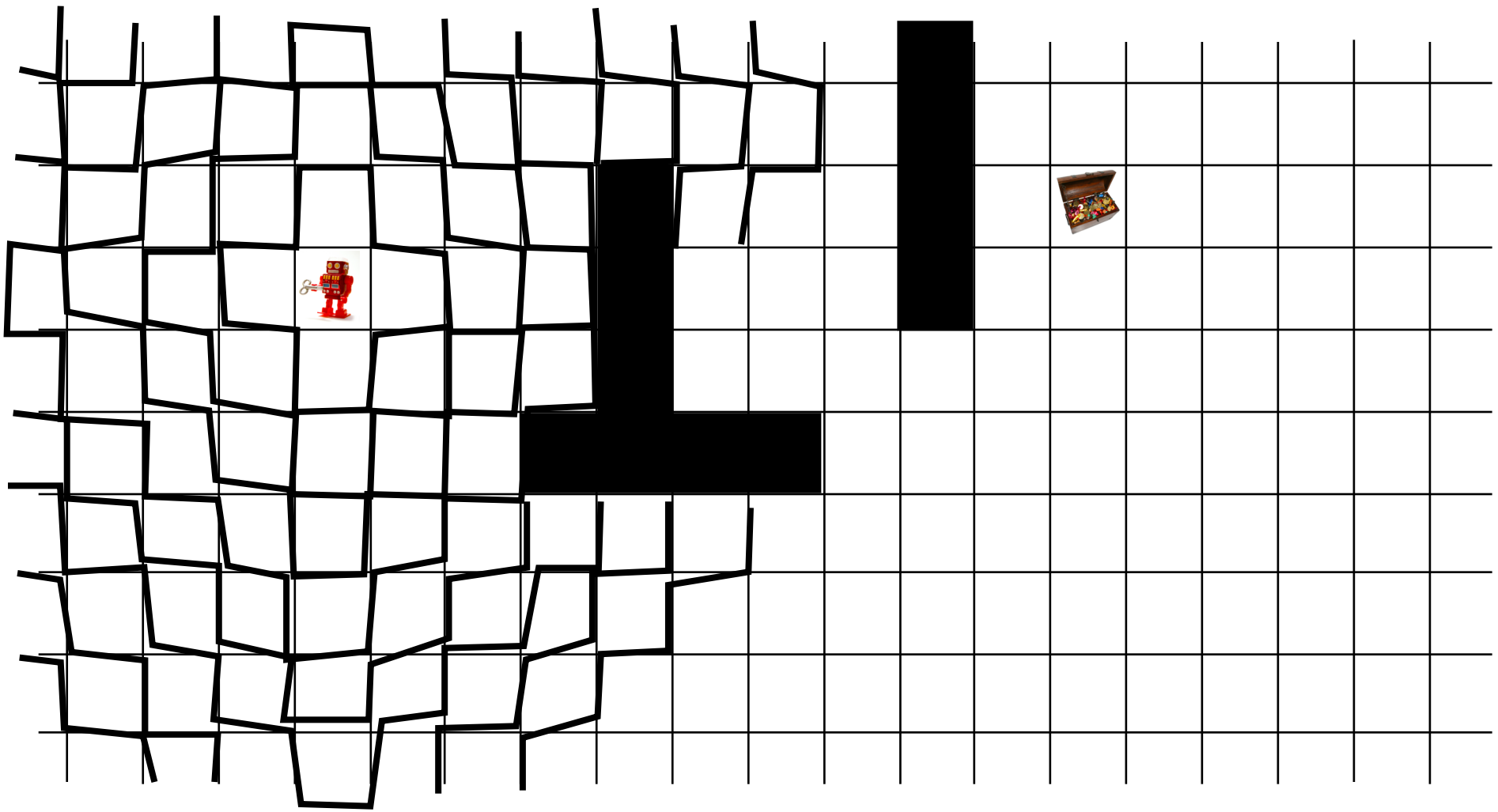
幅優先探索（Dijkstra法）



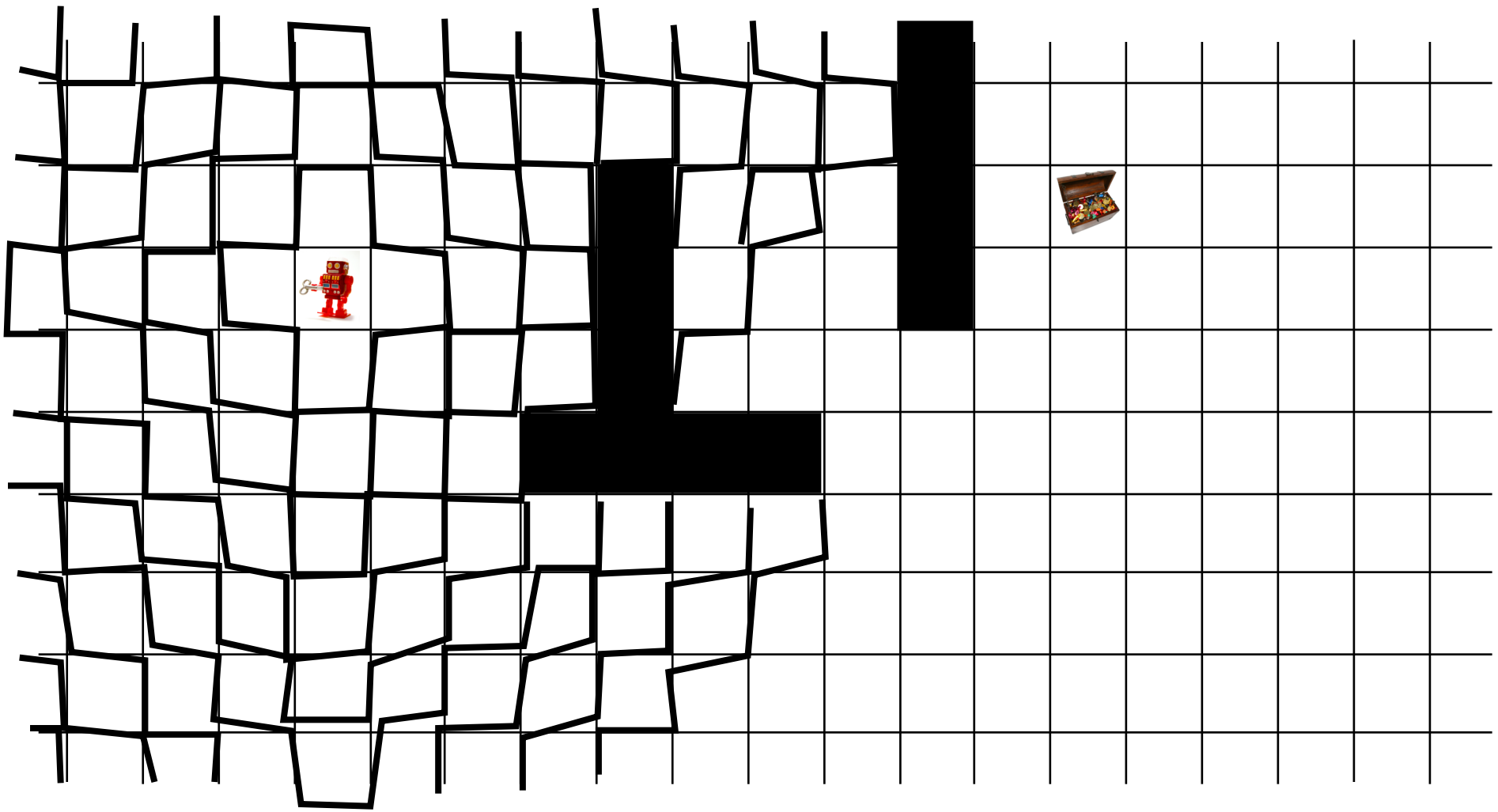
幅優先探索（Dijkstra法）



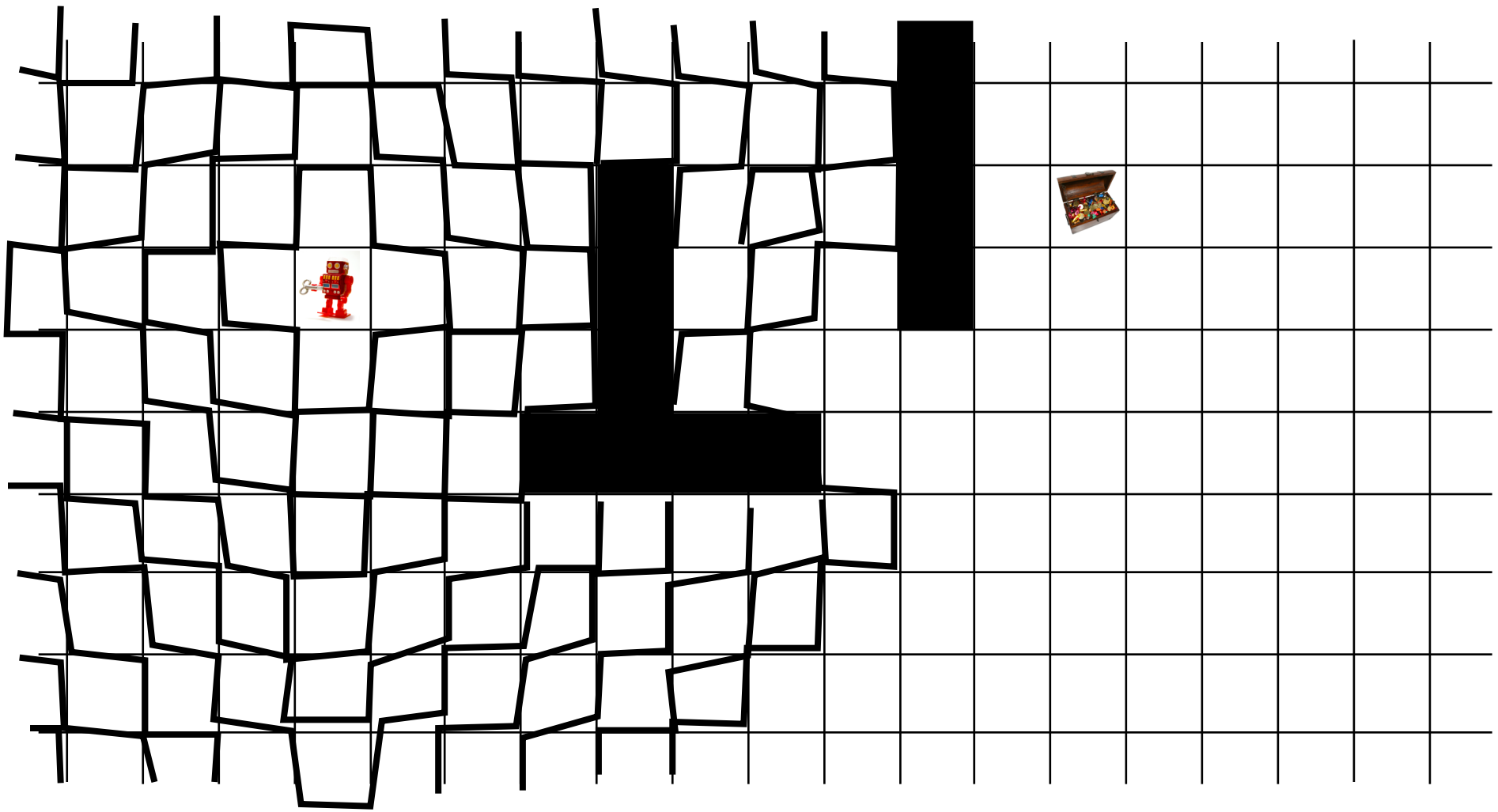
幅優先探索（Dijkstra法）



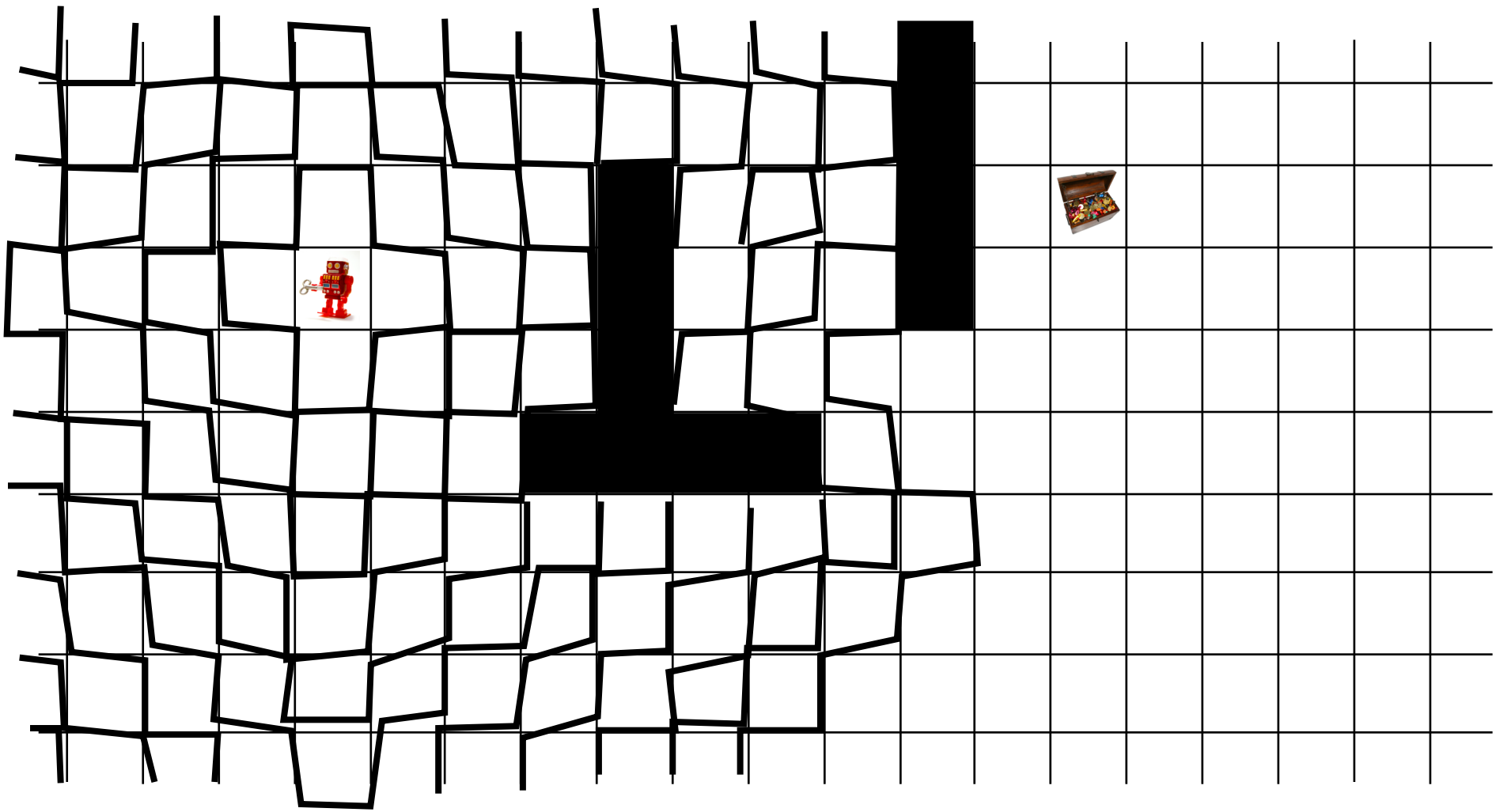
幅優先探索（Dijkstra法）



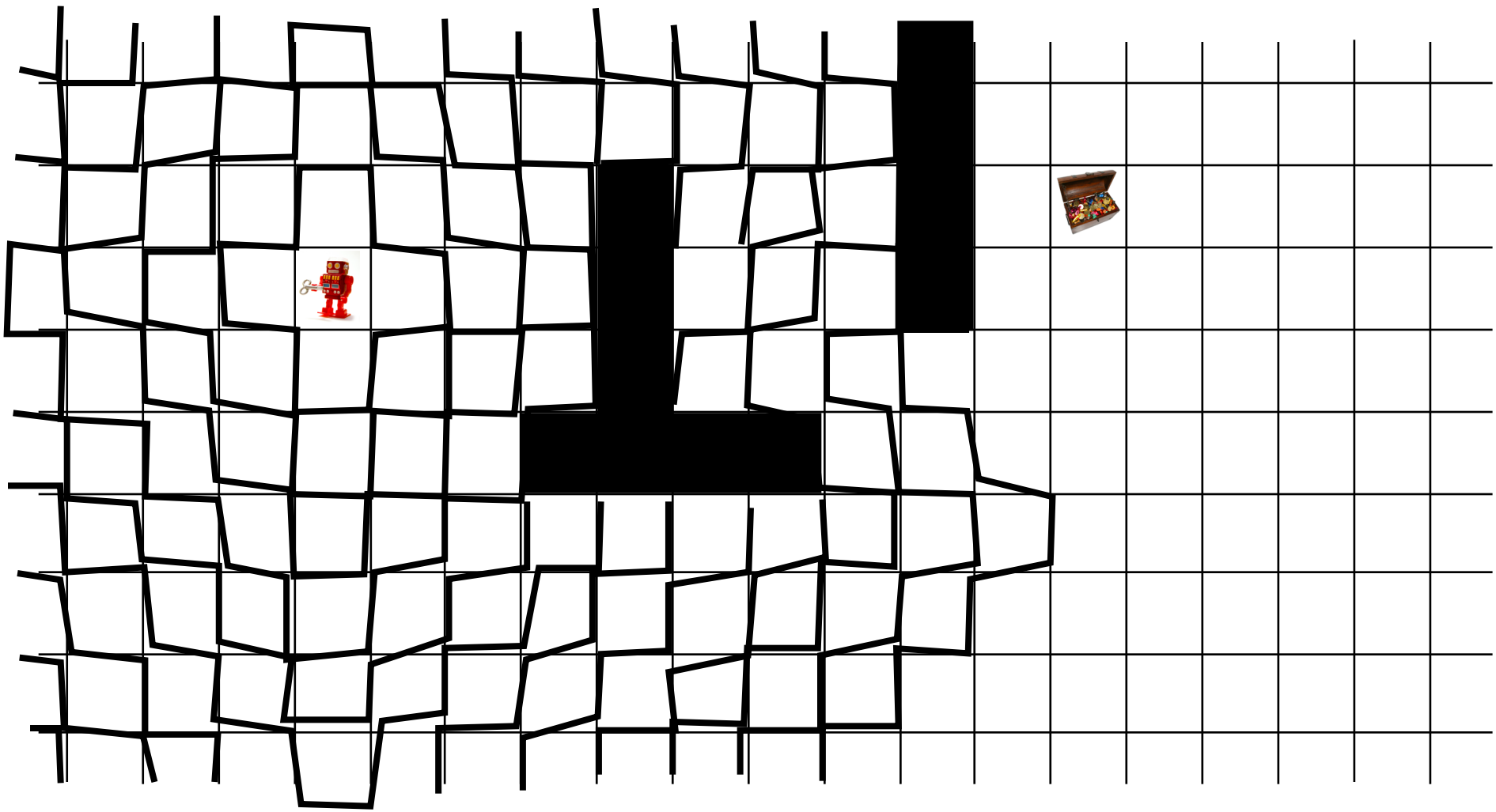
幅優先探索（Dijkstra法）



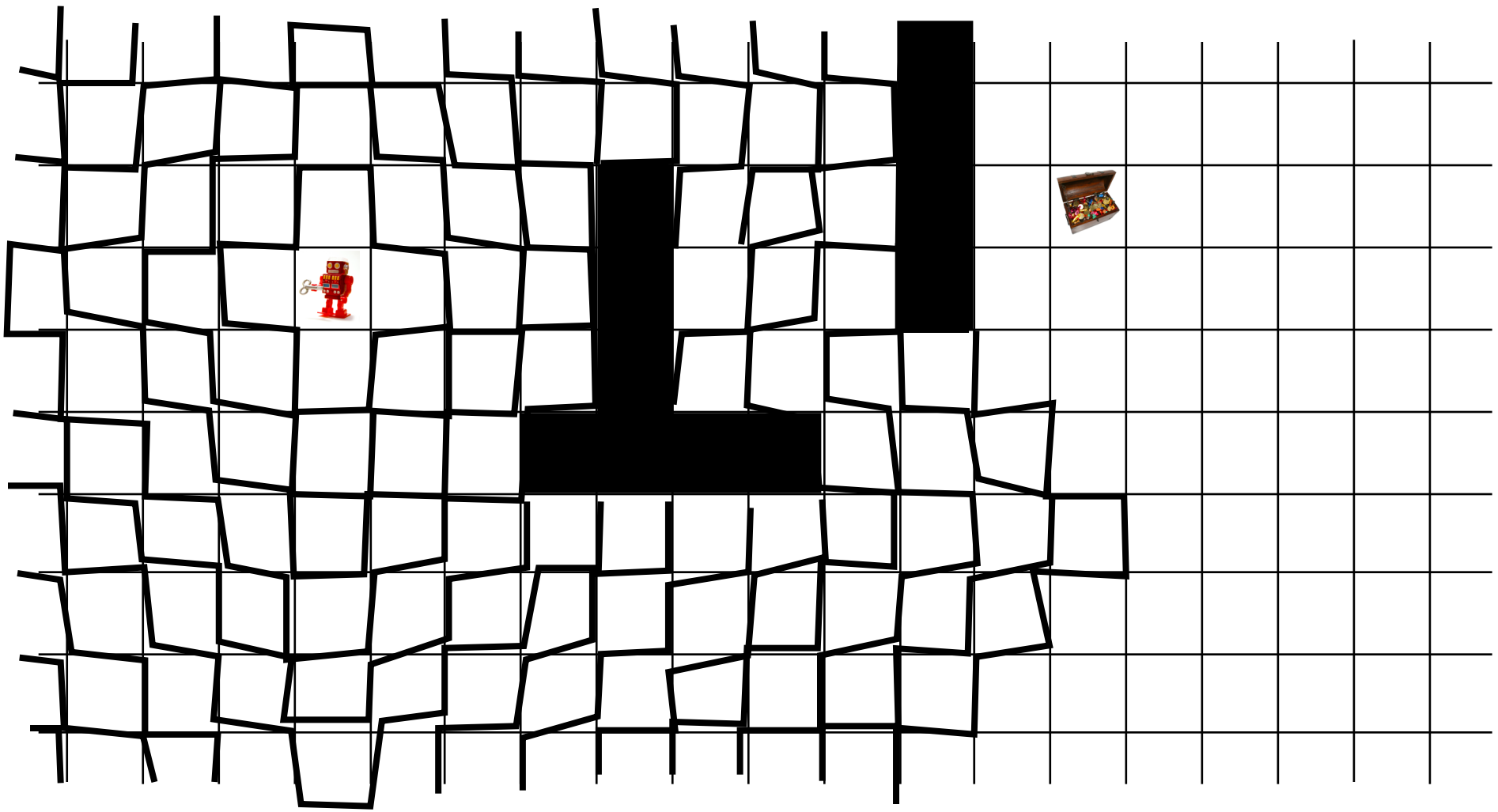
幅優先探索（Dijkstra法）



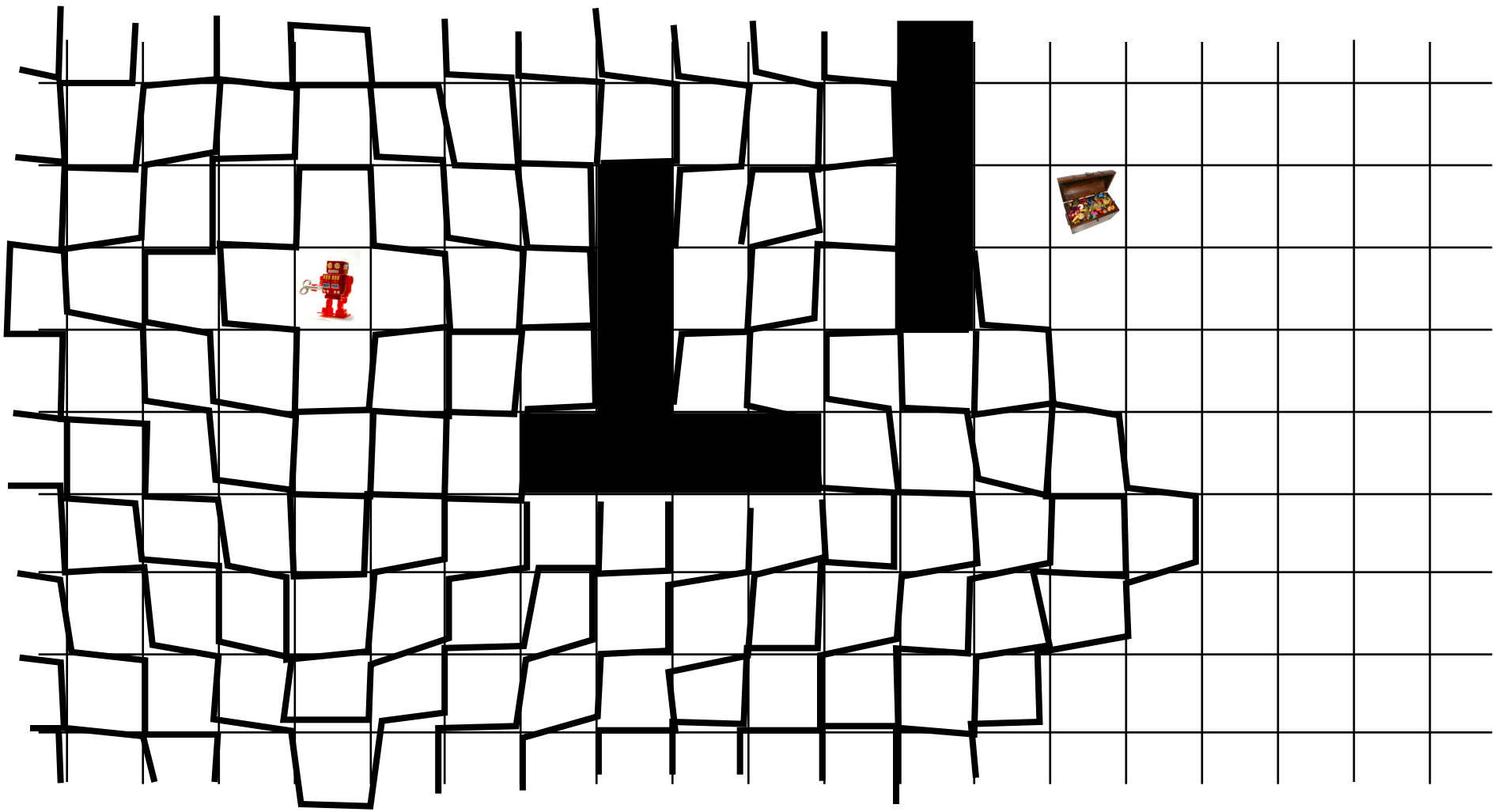
幅優先探索（Dijkstra法）



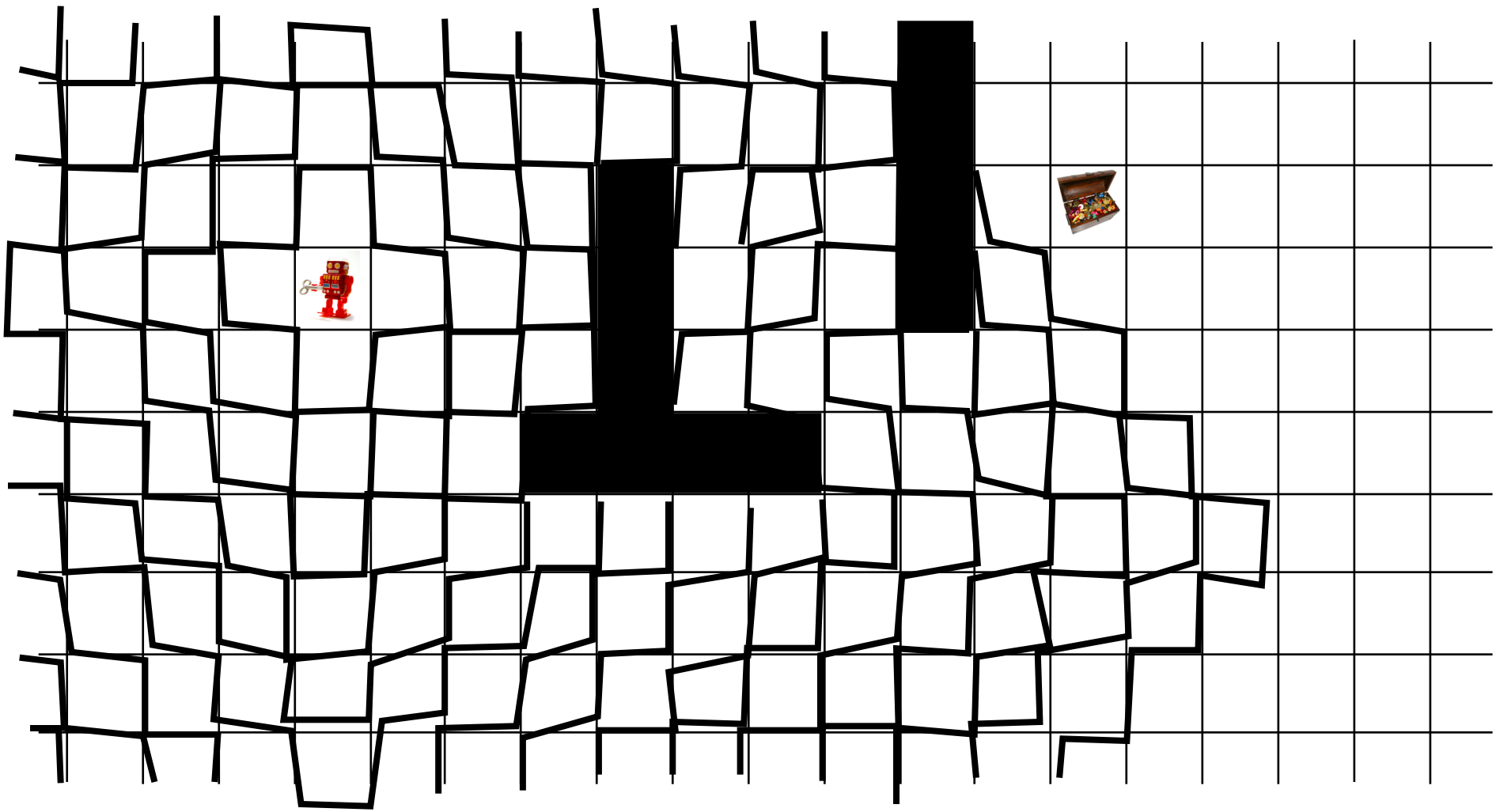
幅優先探索（Dijkstra法）



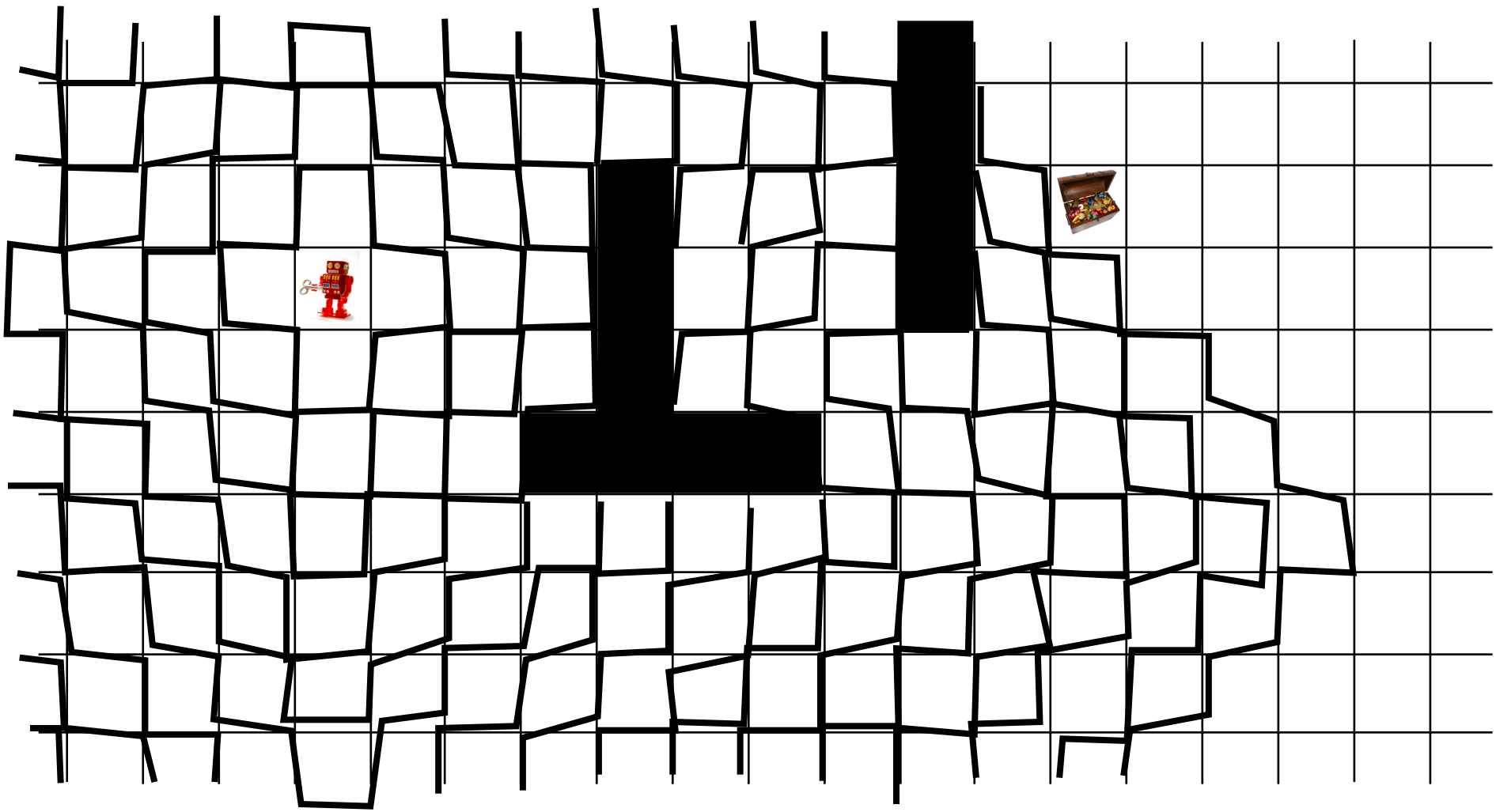
幅優先探索（Dijkstra法）



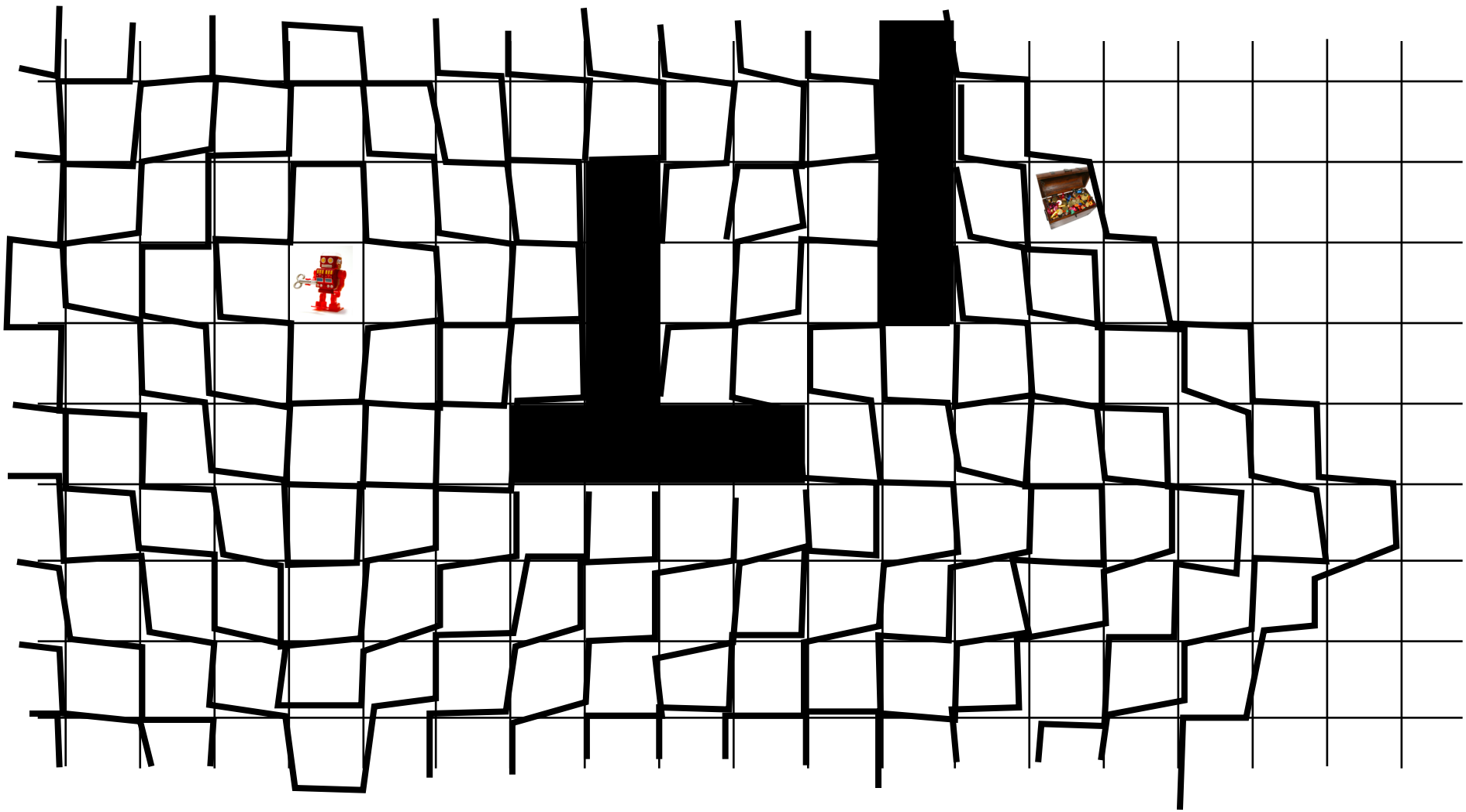
幅優先探索（Dijkstra法）



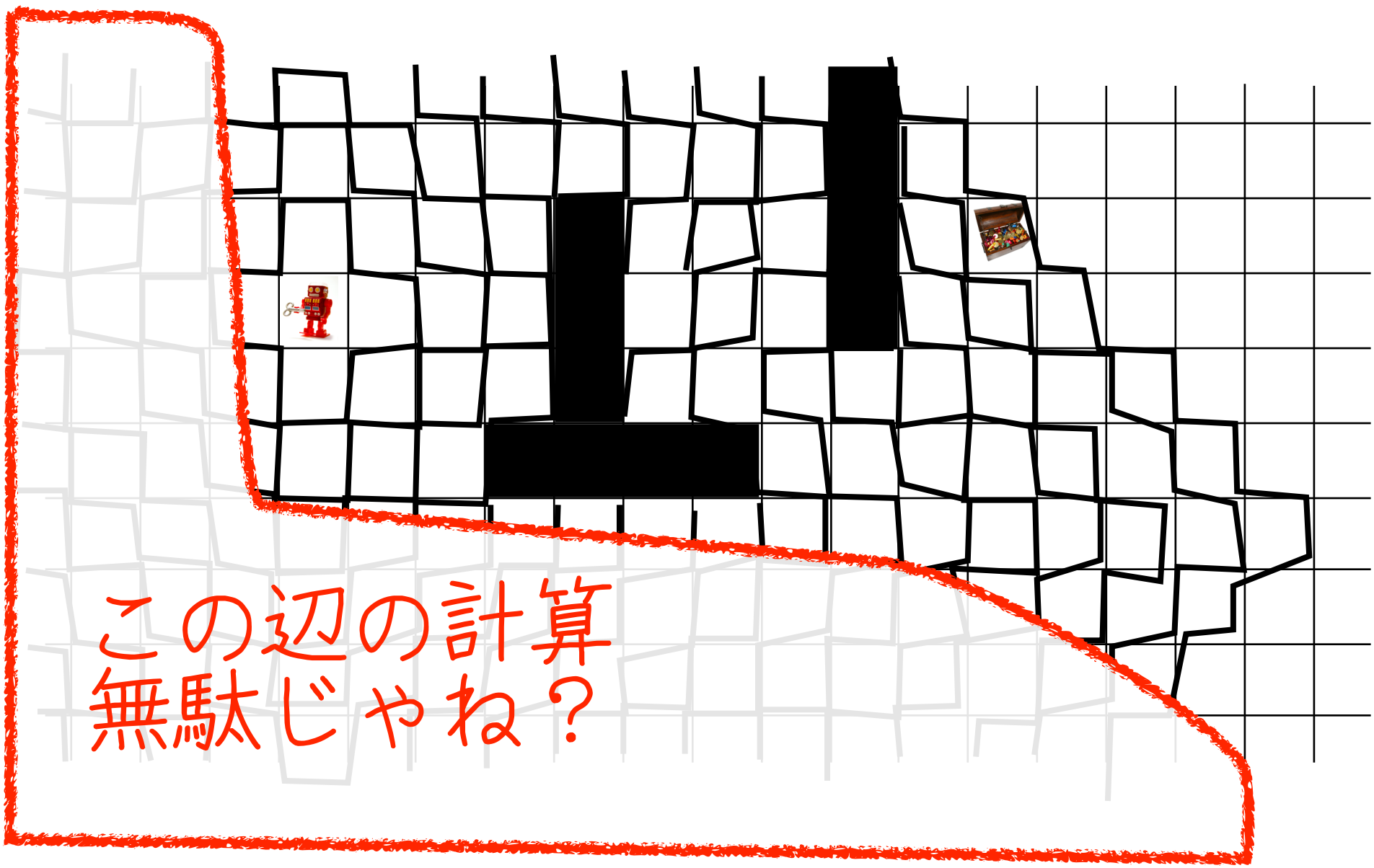
幅優先探索（Dijkstra法）



幅優先探索（Dijkstra法）



幅優先探索（Dijkstra法）



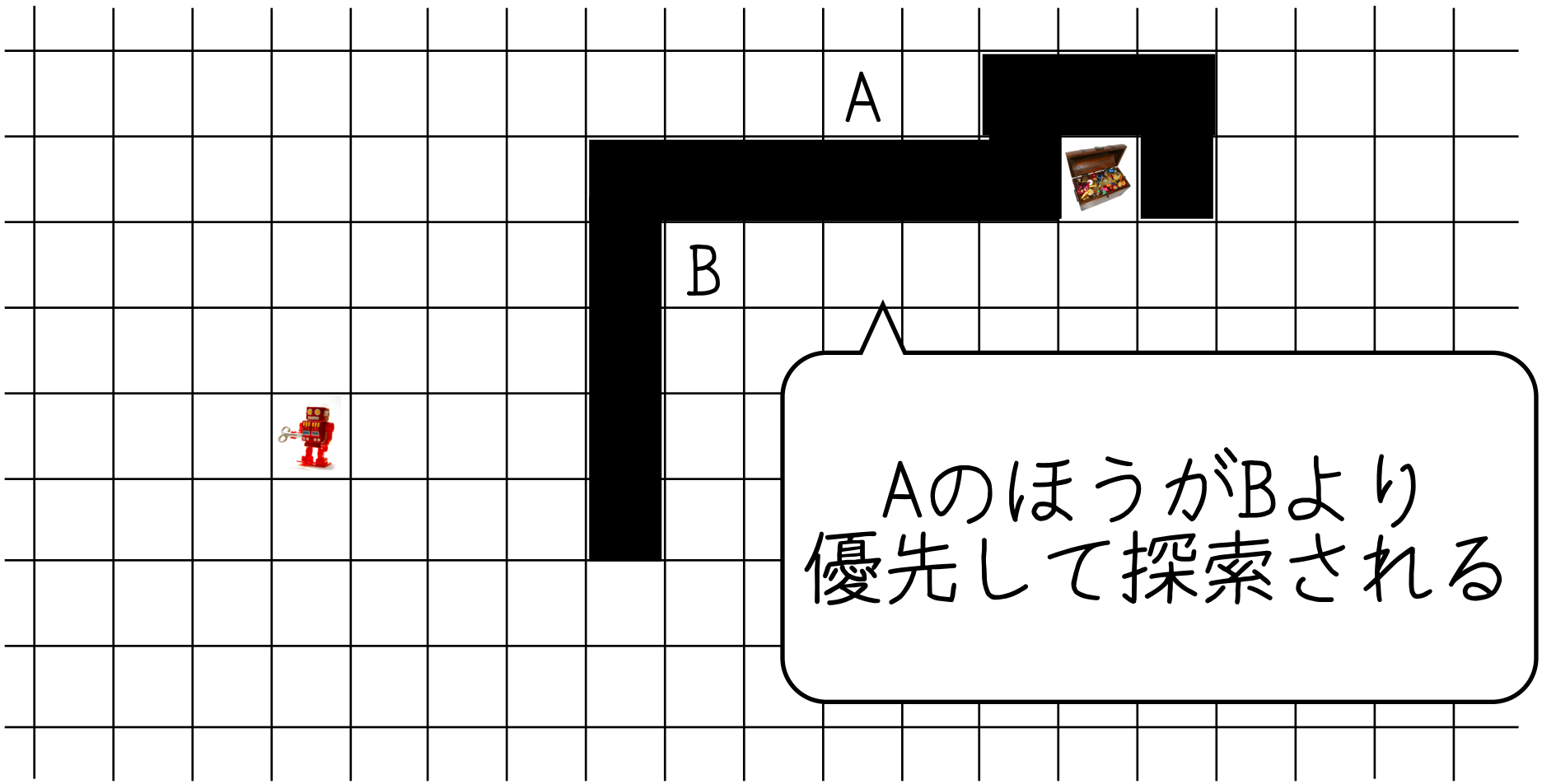
アプローチ1

- ゴールまで「近そう」なやつから優先的に探索
  （最良優先探索）
  - 「近そう」：
    障害物がなかった場合の距離
    * xy座標の差の絶対値の和

- 幅優先でキューのかわりに
  優先度付きキューを使う

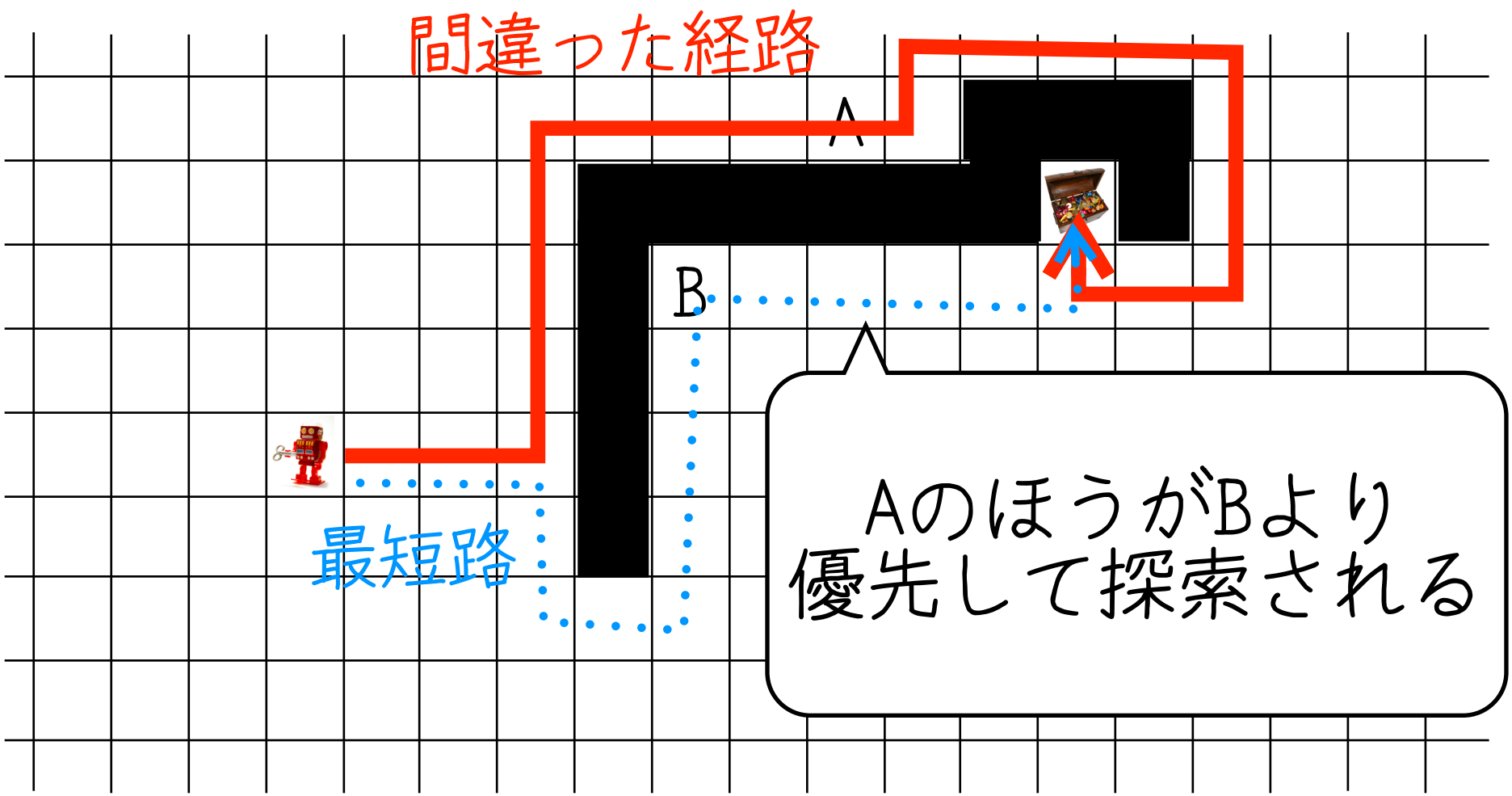
問題点

- 最短性が失われる場合がある



問題点

- 最短性が失われる場合がある



アプローチ2

- 「これまでの距離」＋
  「ゴールまでの距離の見積り」
  が小さいものを優先して探索
  - ゴールまでの距離の見積り：
    障害物がなかった場合の距離

  - 実はこの場合は
    最短経路がきちんと求まる

A*探索

しかしh*は未知
(gは探索の過程でわかる)

f*(x) = g(x) + h*(x)
g: ロボからの距離
h*: 宝への距離
が小さいxを優先し選ぶのがベスト

x

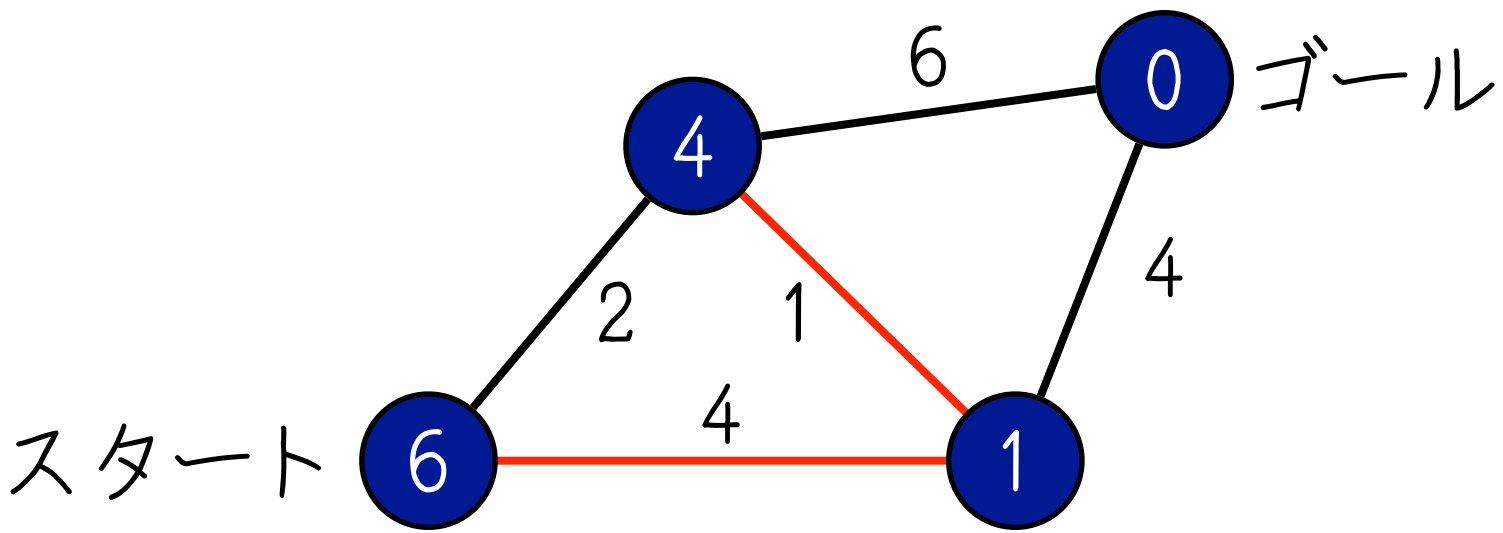
A*探索はf(x)=g(x)+h(x)が小さいxを優先.
但しhは0 ≦ h(x) ≦ h*(x)を満たす
(admissible)

このとき解の最短性が保証

注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

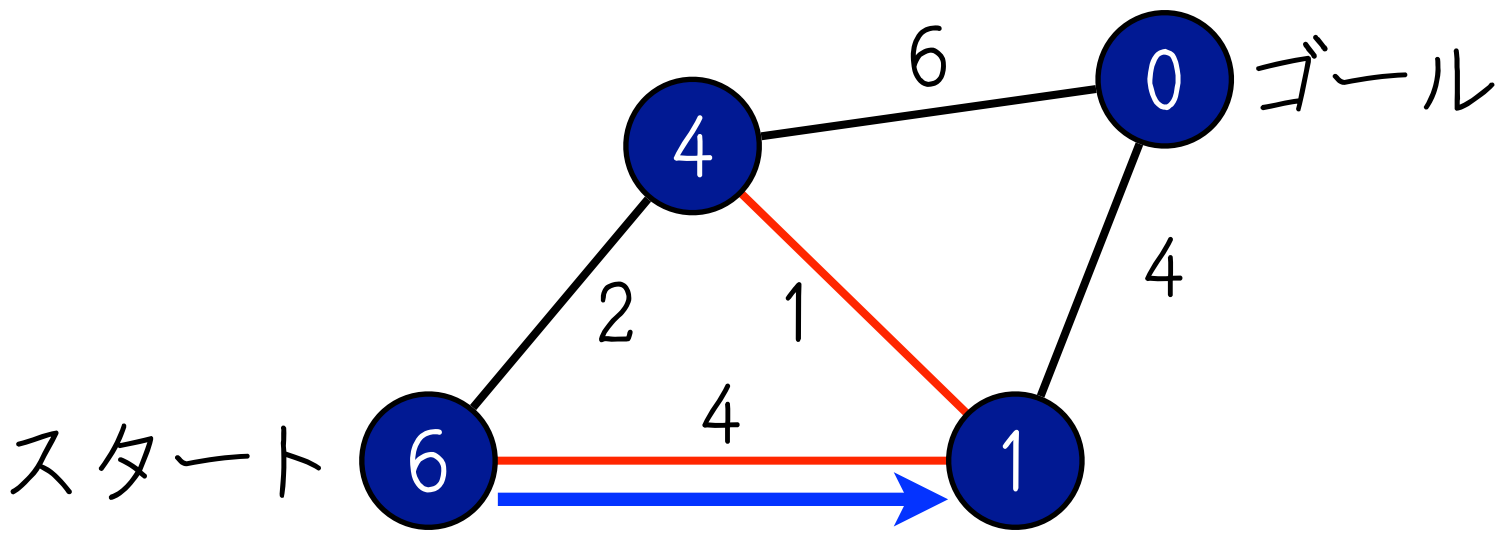
- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

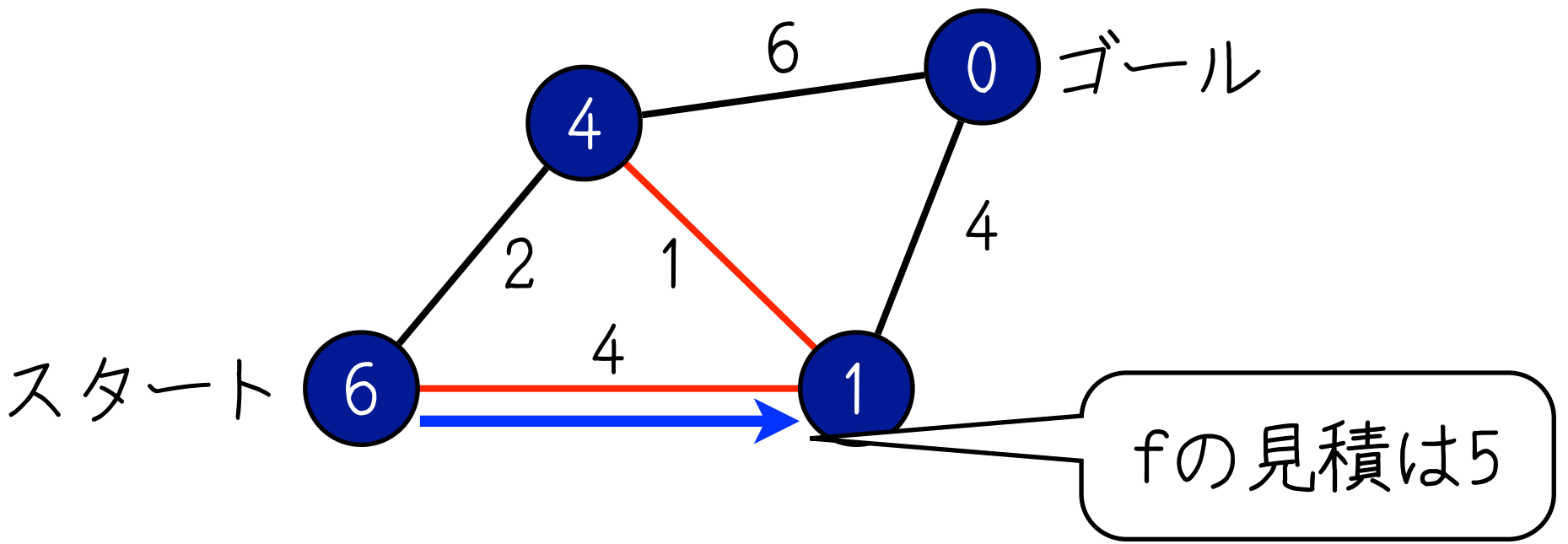
- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

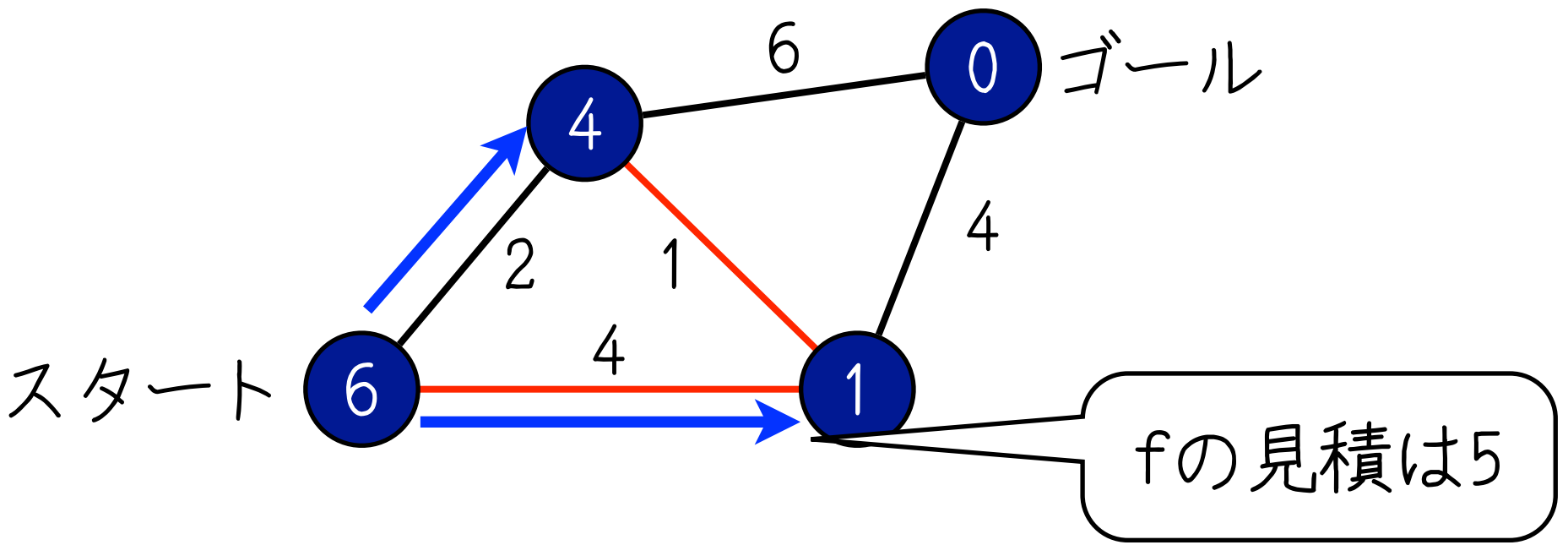
- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

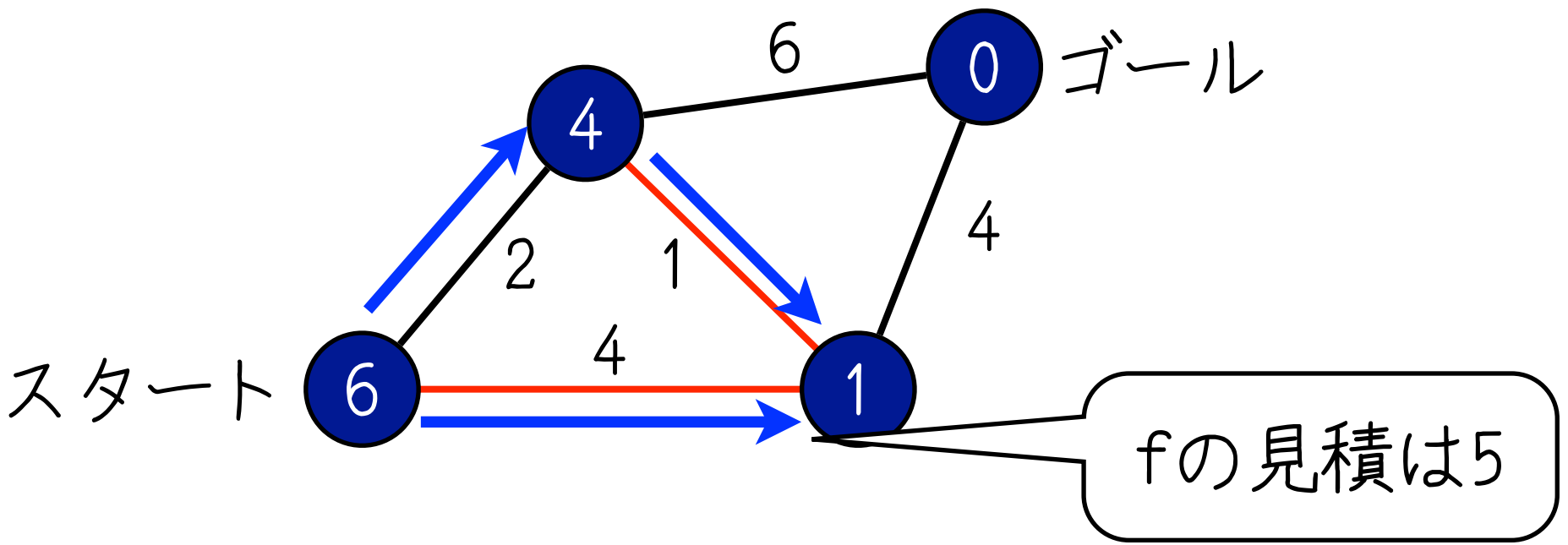
- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

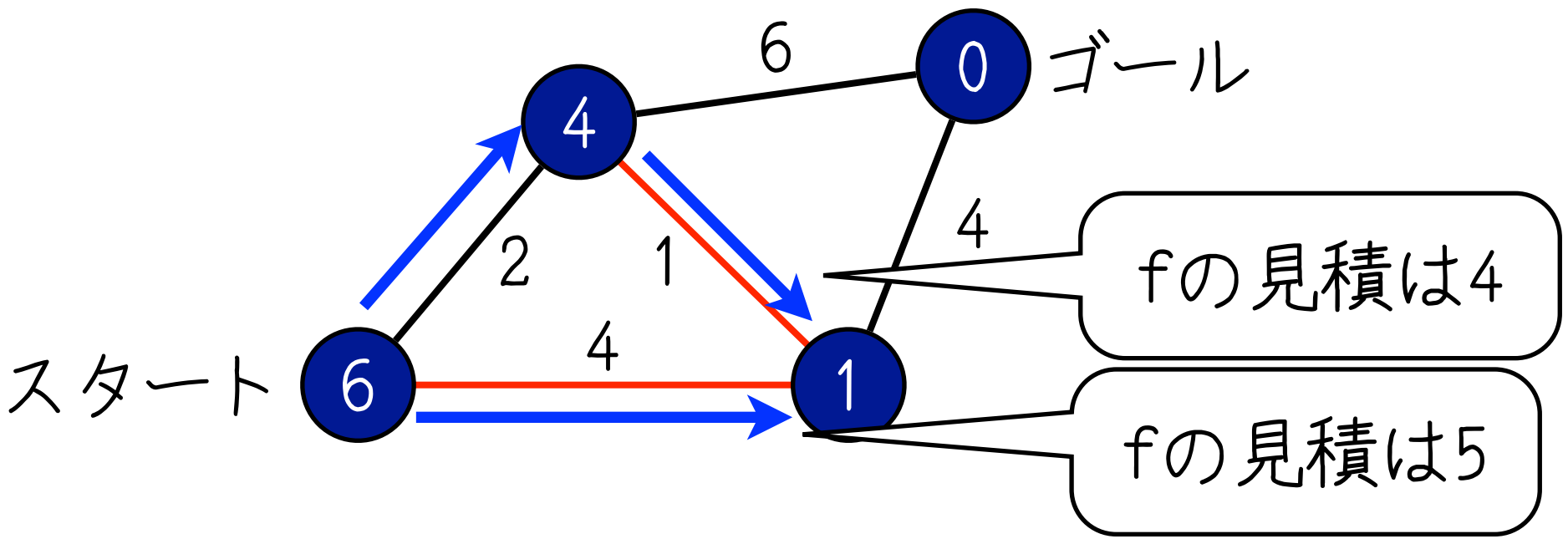
- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



注意

f(x) = g(x) + h(x)
g:スタートからxまでの距離

- 一般には同じ接点を何度も巡る必要
  - 再び巡ったほうがfが小さくなる
  - そうする必要がないための条件
    $|h(x)-h(y)| \leqq dist(x, y)$



IDA*

- A*に反復深化のアイデアを適用
  - 途中で打ち切る深さ優先探索を繰り返す
  - ただし，深さの代わりにfを打ち切りに利用する
    - 最初はlimit = f(初期状態)
    - f(x) > limitな分岐は打ち切る
    - 打ち切られたノードxについて，最小のf(x)を次のlimitとする

まとめ

- いろいろな探索手法を紹介
  - 深さ優先探索（Depth-First Search）
  - 幅優先探索（Breadth-First Search）
  - 反復深化（Iterative Deepening）
  - A*

第13回レポート課題
締切 7/22 13:00

注意

- 今日の課題はPrologで解かなくてもよい
  - HaskellやOCamlでも可
  - もちろん，Prologで解いてもよい
    - が何かうれしいことがあるわけではない
  - 他の言語はダメ
    - この講義は「関数・論理型プログラミング実験」

問1

- 探索木のデータを適当に与え，DFS，BFS，反復深化を実装し比較せよ
  - 無限に深い部分木がある場合も考察せよ

Haskellの場合

```
data SearchT a
     = SNone | SUnit a
     | SOr (SearchT a) (SearchT a)
```

(OCamlの場合は無限木を表現するために一工夫必要)

問2

- 問1の探索木を用いて
  全ての「自然数の有限リスト」
  を列挙する処理を実装せよ
  - ヒント
    - SearchT [Int]を返す関数を実装
      - Haskellなら
        SearchTがMonadPlusのインスタンスと
        できることを利用する
        - 第8回問2も参考
        - 直接「全ての『自然数の有限リスト』
          からなる無限リスト」を返そうとする
          と大変である．余裕があれば確認せよ

問3

- 例で出したロボの移動の問題について様々の探索手法を実装し比較を行え
  - ロボの位置，ゴールの位置は座標の組で与えよ
  - 障害物の位置は座標の組のリストで与えよ
  - ロボは上下左右に1マスずつ移動できる
  - 少なくとも，以下の2つは実装すること
    - A*
    - A*，IDA*以外の探索法

問4

- 問3で
  - ロボの移動を上下左右斜め8方向にした場合（王将と同じ動き）
  - ロボの移動を桂馬とび（8方向）にした場合
    - A*のhが「ゴールまでの座標差」ではダメなことに注意

問5

- 問4でロボが王将の動きをする場合，探索手法によっては，ロボが無駄に斜めに移動する場合がある．これを防ぐにはどうすればよいか議論せよ．

発展

- 15パズルの解答手順を求めるプログラムを書け
  - 様々な探索アルゴリズムを使用し比較せよ

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

次回はCurry

- 講義ではMCCを使う
  - http://danae.uni-muenster.de/~lux/curry/
    - 使えるようにしておくこと
      - 以下で手にはいる版を使う

```
darcs get --lazy \
  http://danae.uni-muenster.de/~lux/curry/darcs/curry
```

        - noweb等必要なものは適宜インストールせよ
  - 他の処理系
    - http://www-ps.informatik.uni-kiel.de/currywiki/implementations/overview