目次

Ⅰ．概要

空間インデックスとは

空間上に 位置・大きさ といったデータを持って存在する

オブジェクト (空間オブジェクト) の集合の中から

特定のオブジェクトの抽出・操作を行うアルゴリズム

矩形 A
座標 X : 5.3, 座標 Y : 11.1
幅 W : 27.9, 高さ H : 19.0

今回のアルゴリズム

空間オブジェクトの N 個集合と同じ空間内に存在する

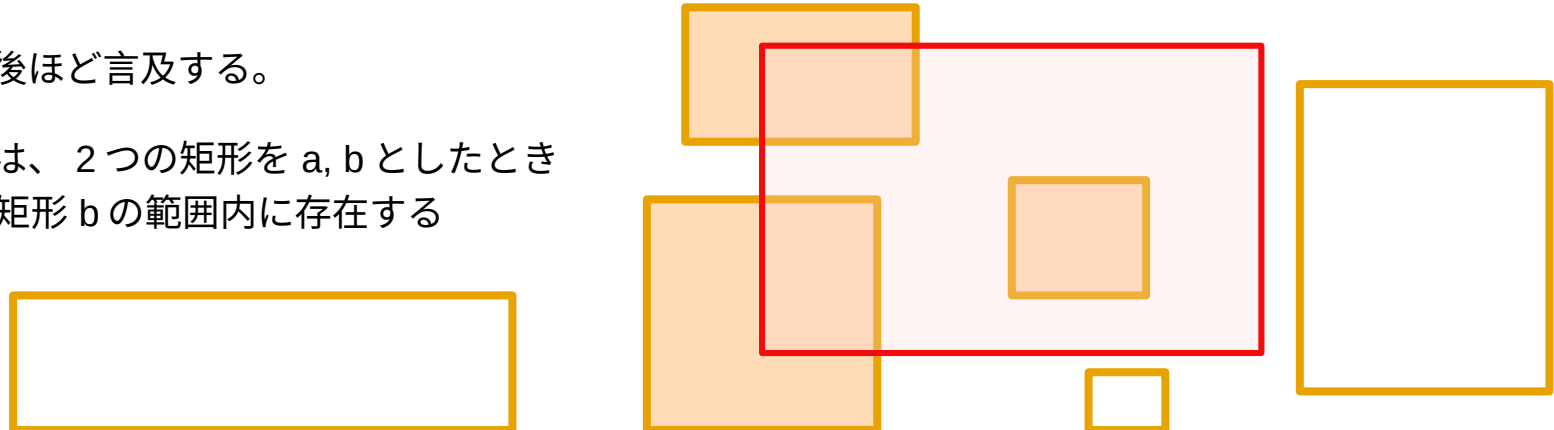
特定の一つの矩形範囲 (抽出範囲) について、それと一部 or 全体が

重なっているオブジェクトを全て抽出するアルゴリズム (範囲質問)

(なお、本スライドの説明では、基本的に 2 次元の座標系を用い 、空間オブジェクトはそれぞれ
XY 座標、幅、高さ (x, y, w, h) を持った矩形で表せるとする。

その他の次元については、後ほど言及する。

また、「重なっている」とは、 2 つの矩形を a, b としたとき
矩形 a のいずれか一頂点が矩形 b の範囲内に存在する
状態を指す。



今回のアルゴリズム

抽出範囲が時間等に応じて変化・更新される場合において、

従来の手法に比べてより最適な計算量を実現

注意
まだ研究段階で、上記は最終的な目標。現時点では、未だ部分的な実現にとどまっている。
(現状に関しては、「 V. 現状と課題」で説明)

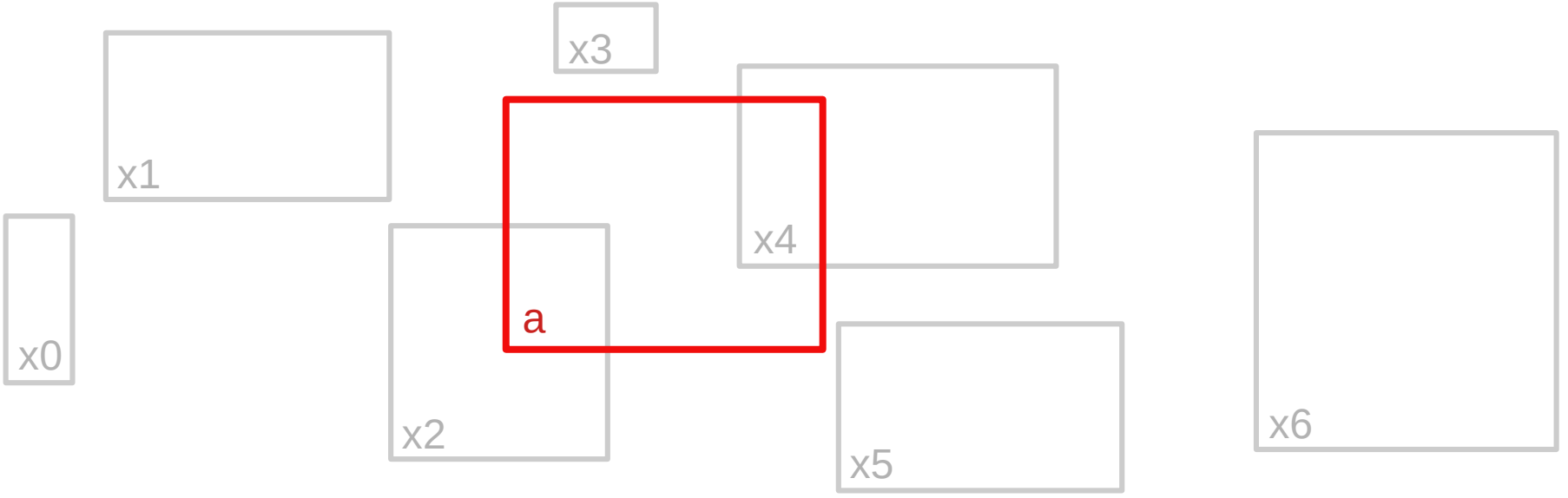
また、オブジェクトの位置や大きさに変化が現れない
static な空間におけるシチュエーションは想定しておらず、そのときは
計算量は愚直なケースとほぼ変わらない (R-tree で代用しよう)

II．アルゴリズムの内容

この項以降は”空間オブジェクト”を
“ 空間 obj”と省略します 許して

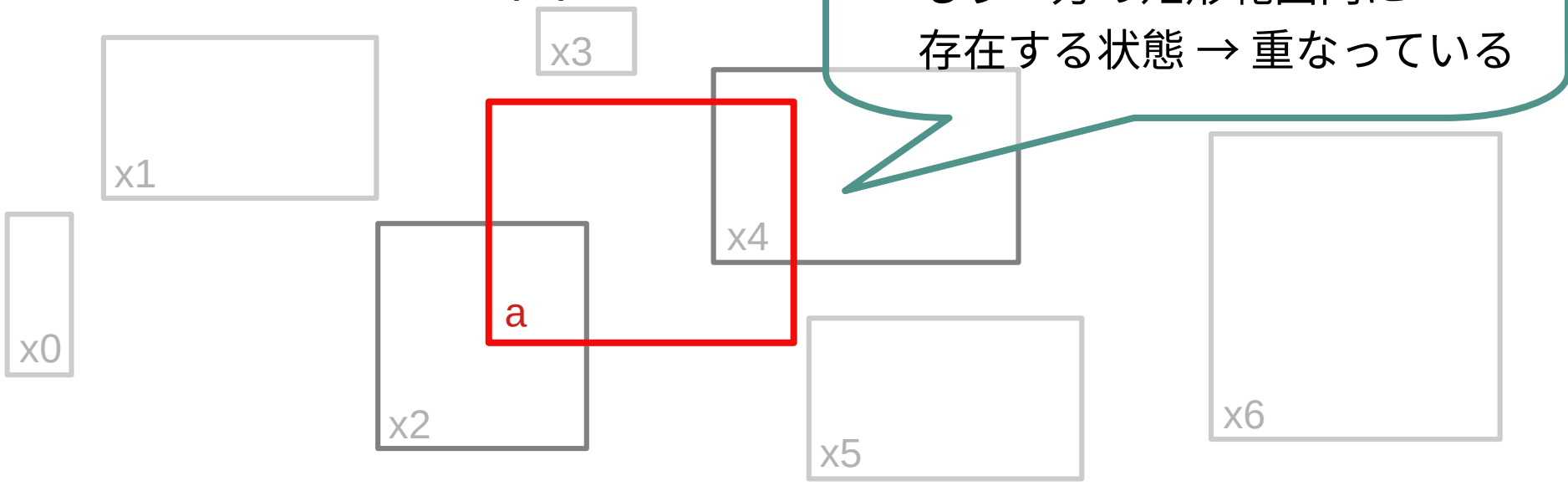
このようなシチュエーションを考える

- 矩形が N 個あると仮定し
  図の矩形範囲 : a (赤枠) と重なっている
  空間 obj 集合 : X (灰色枠) 中の 空間 obj を全て抽出する。



愚直な実装

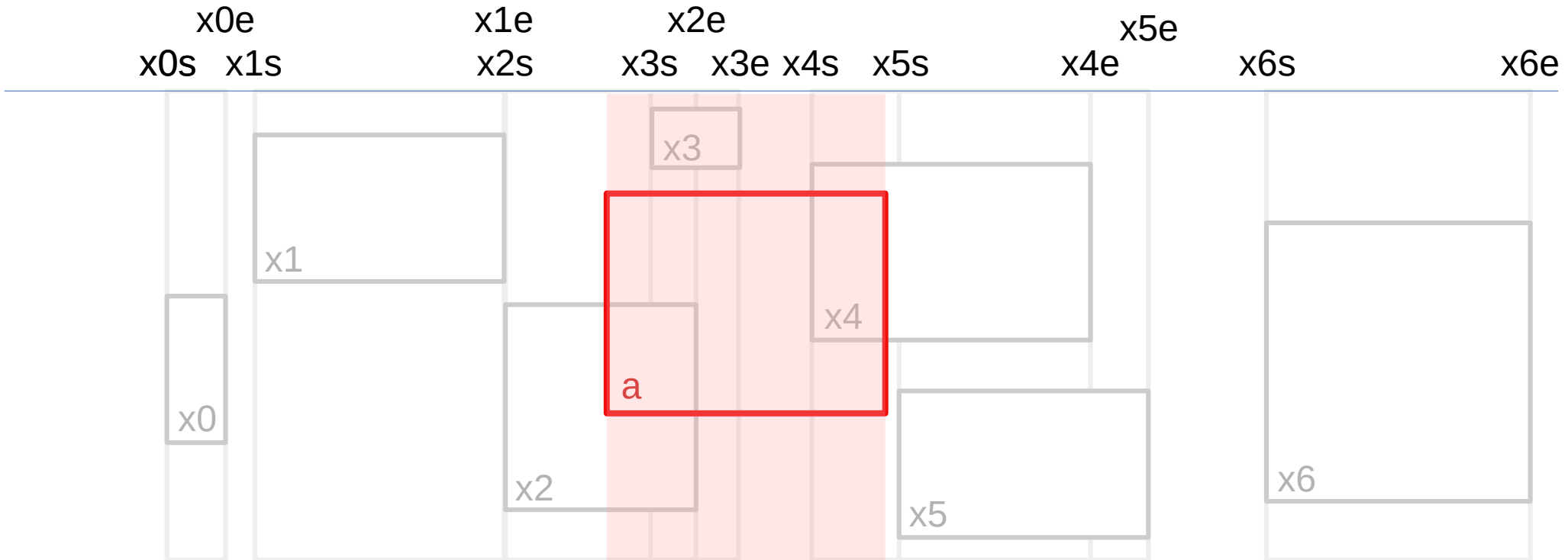
- 全ての X の要素に対して a との重なり判定を行う
  (条件式に当てはめるだけ)
- 計算量はもちろん、 O(N)



本アルゴリズムによる実装

- まずは、 x 座標範囲と重なっている X 要素の集合を抽出する

図の赤枠



本アルゴリズムによる実装

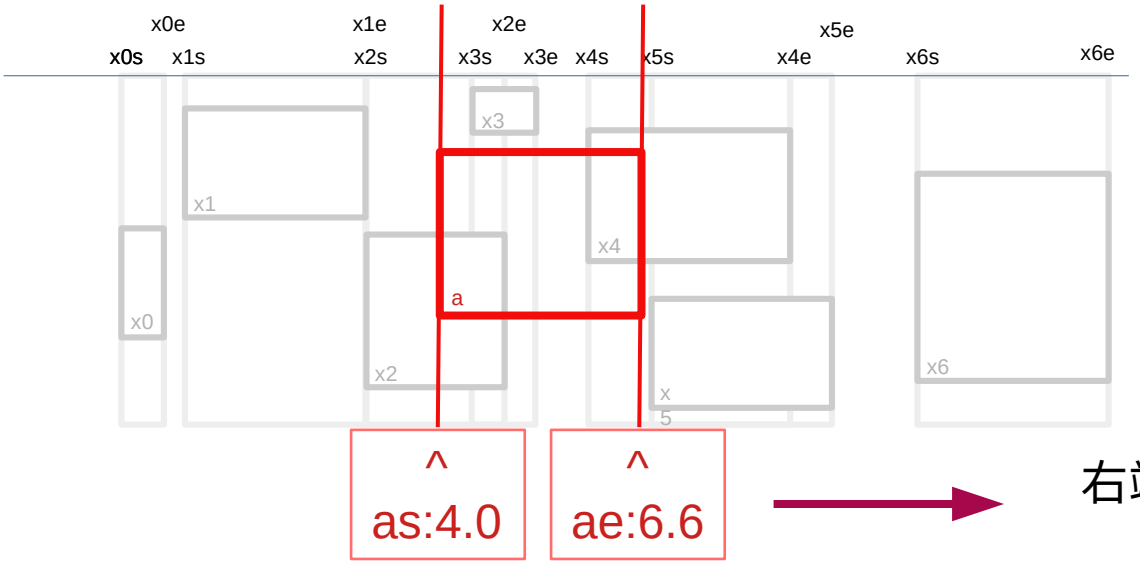
- X 要素の左端 / 右端の x 座標をメモ、連想配列 (等) にまとめる
  (連想配列内の要素は x 座標でソートされる)

	x0	x1	x2	x3	x4	x5	x6
左端	0.0	0.8	3.0	4.5	6.2	6.7	8.7

	x0	x1	x2	x3	x4	x5	x6
右端	0.6	3.0	5.0	5.5	8.0	8.3	9.9

本アルゴリズムによる実装

- 矩形 a の左端 / 右端の x 座標をメモ。連想配列には挿入せず、
  ポインタとして扱う (左端ポインタは右端配列に、右端ポインタは左端配列に！)



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

^
ae:6.6

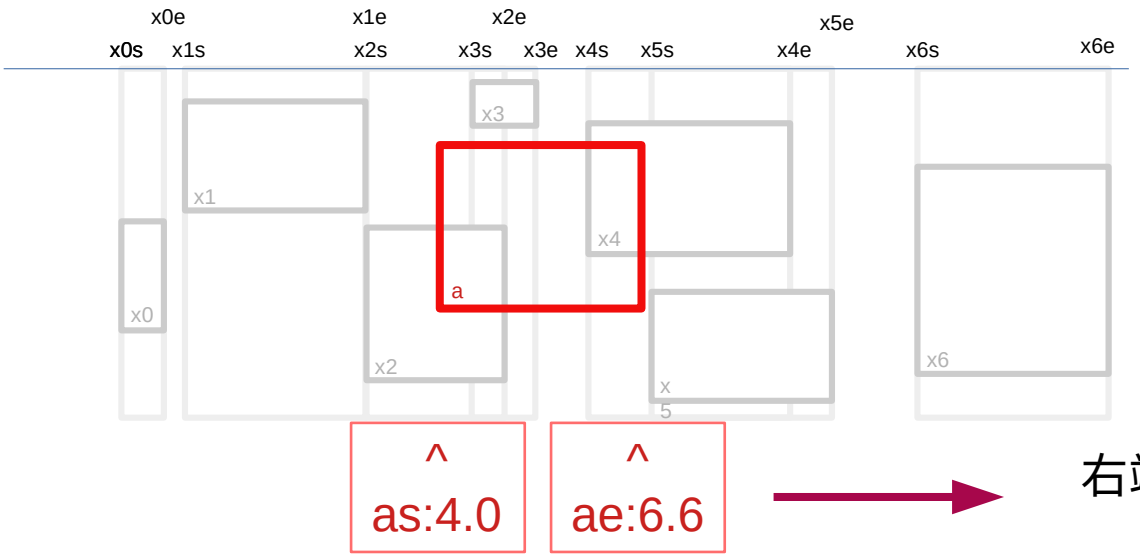
右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

^
as:4.0

本アルゴリズムによる実装

- 何が起こる？→「a の右端より左側の範囲」と重なっている X の要素と
  「a の左端より右側の範囲」と重なっている X の要素を調べることができる



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

^
ae:6.6

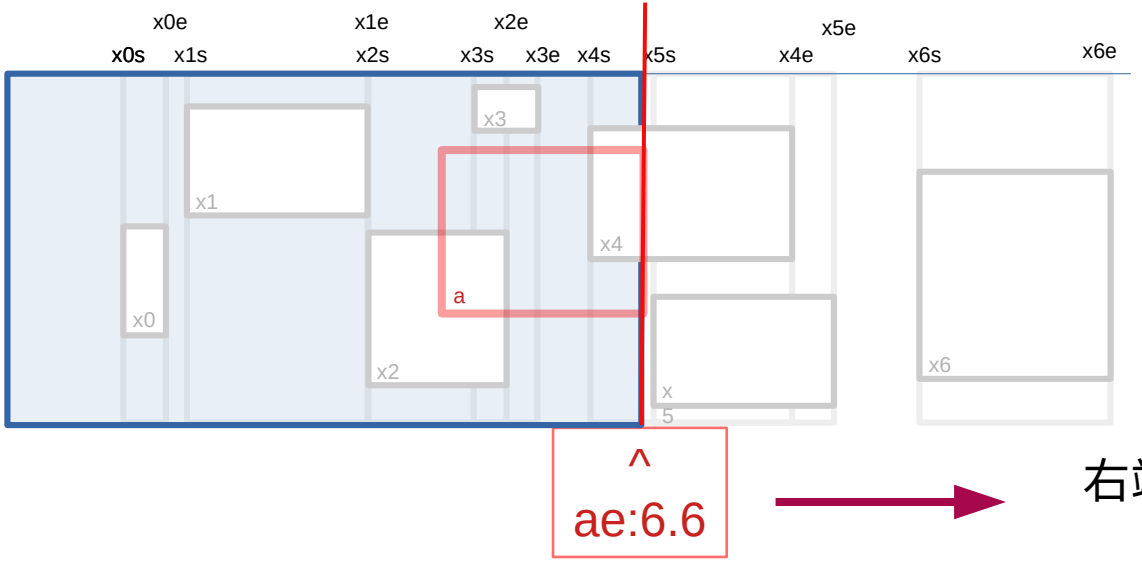
右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

^
as:4.0

本アルゴリズムによる実装

- 何が起こる？→「a の右端より左側の範囲」と重なっている X の要素と
  「a の左端より右側の範囲」と重なっている X の要素を調べることができる



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

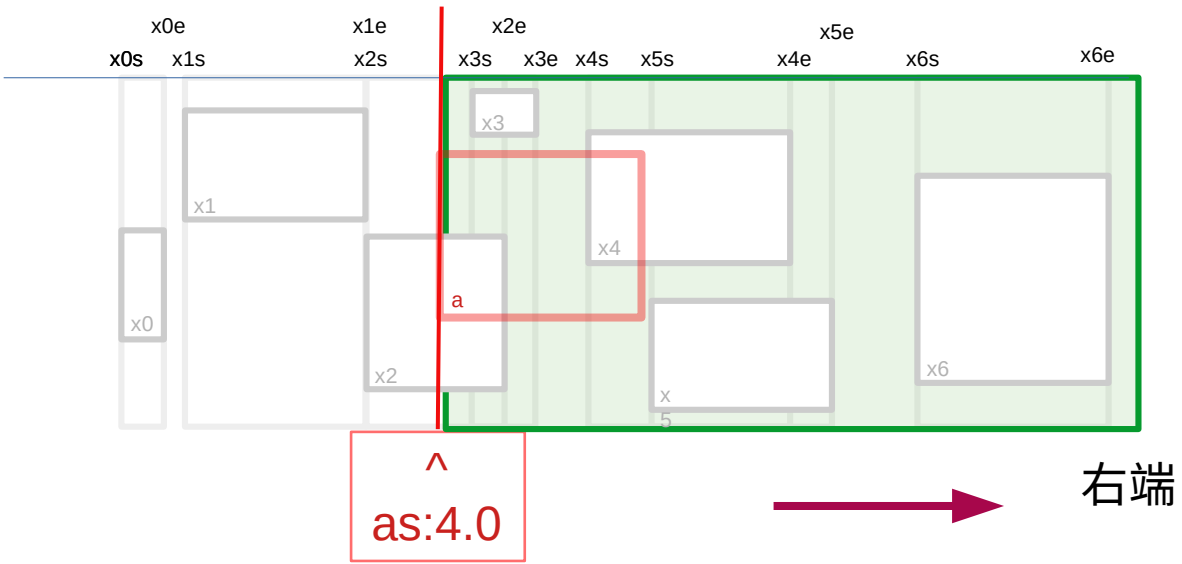
^
ae:6.6

右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

本アルゴリズムによる実装

- 何が起こる？→「a の右端より左側の範囲」と重なっている X の要素と
  「a の左端より右側の範囲」と重なっている X の要素を調べることができる



	x0	x1	x2	x3	x4	x5	x6
左端	0.0	0.8	3.0	4.5	6.2	6.7	8.7

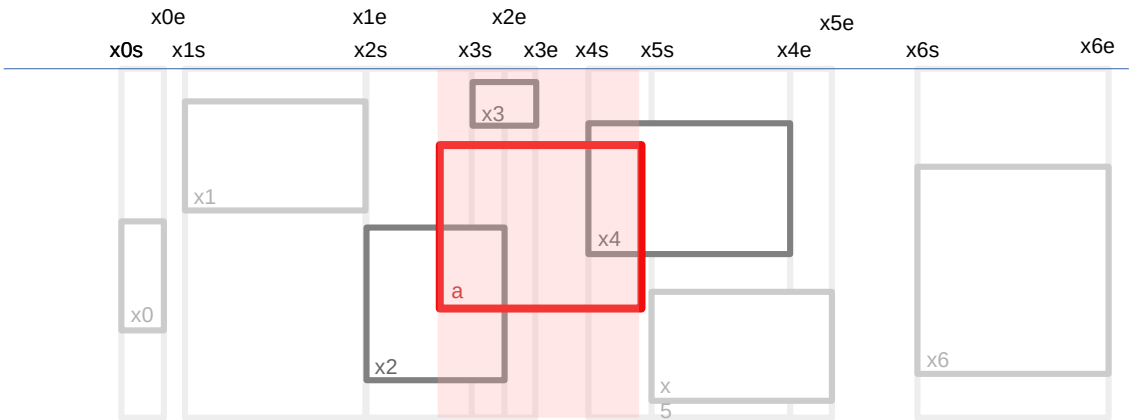
	x0	x1	x2	x3	x4	x5	x6
右端	0.6	3.0	5.0	5.5	8.0	8.3	9.9

^
as:4.0

本アルゴリズムによる実装

- さっき抽出した 2 つのパターンにおける X の要素の集合について、積集合をとる
  → 抽出ができた！ { x2, x3, x4 }

- これらを hash テーブルに一旦格納



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

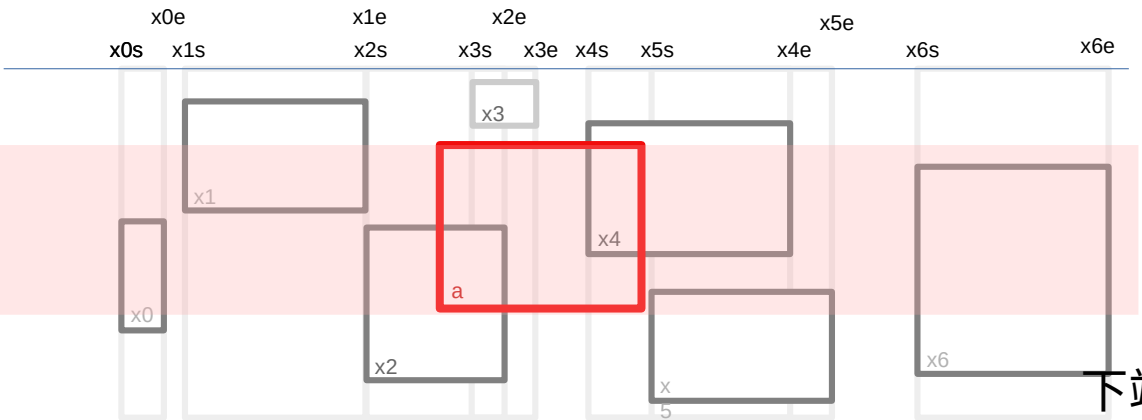
抽出 X (hash_table) :

x2	x3	x4

本アルゴリズムによる実装

- 同じ手順を Y 座標基準でも行い、積集合をとる
  → 抽出ができた！ { x0, x1, x2, x4, x5, x6 }

- これも別の hash テーブルに一旦格納



上端

x3	x2	x4	x6	x0	x2	x5

下端

x3	x2	x4	x6	x0	x2	x5

抽出 X (hash_table) :

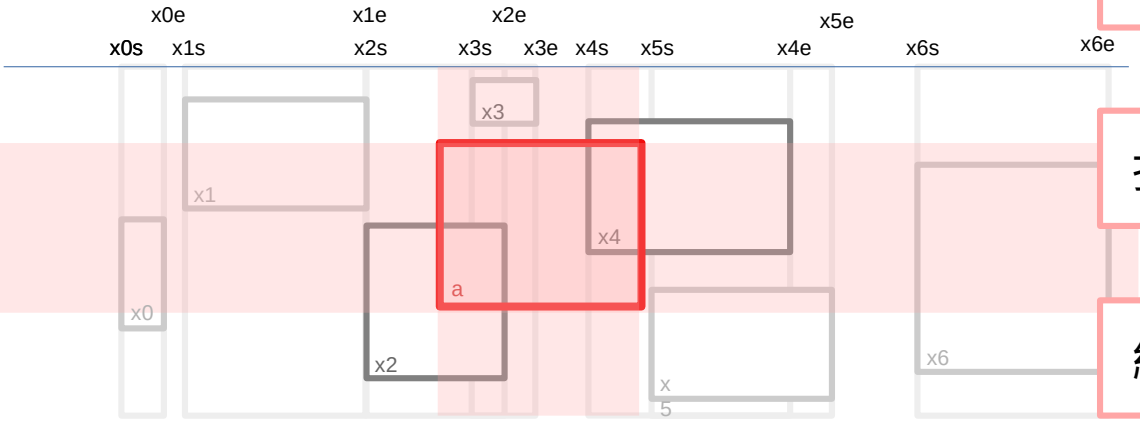
x2	x3	x4

抽出 Y :

x0	x1	x2	x4	x5	x6

本アルゴリズムによる実装

- 2 つの hash テーブルについて、再度積集合をとる
- 目的の抽出ができた!



抽出 X : x2 x3 x4

抽出 Y : x0 x1 x2 x4 x5 x6

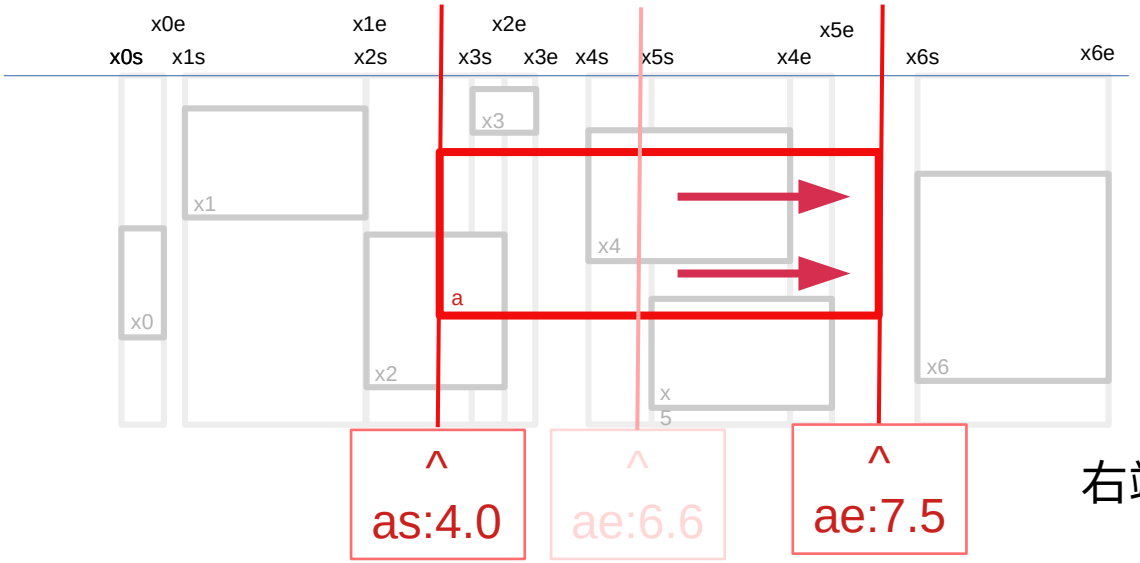
結果 : x2 x4

本アルゴリズムによる実装

- ~~人に説明するってムズカシイ~~
- Two-pointer algorithm(しゃくとり法) からヒントを得た最適化手法
- このとき、時間計算量は O(N) となる
  この時点では、時間計算量は
  愚直実装時との差はない
- しかし、続けて次のシチュエーションを考える
  (ここからが本アルゴリズムの本領)

抽出範囲が動いた時

- 例えば、次の処理にて、抽出範囲の右端が次のように動いたとする



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

^
ae:6.6

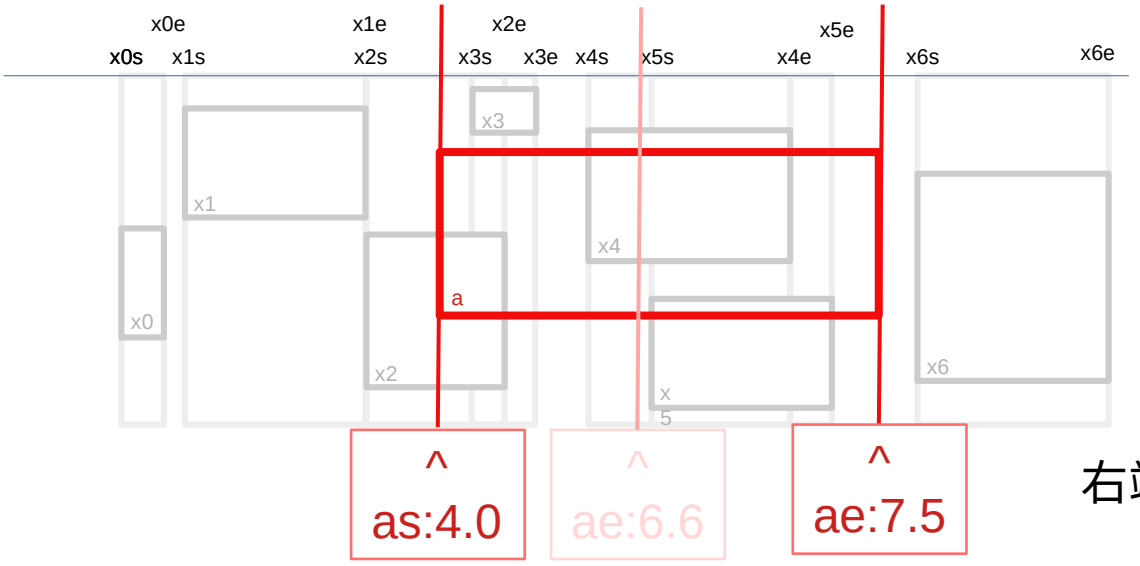
右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

^
as:4.0

抽出範囲が動いた時

- 右端座標の変化に合わせてポインタも動かす



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

ae:6 ae:7.5

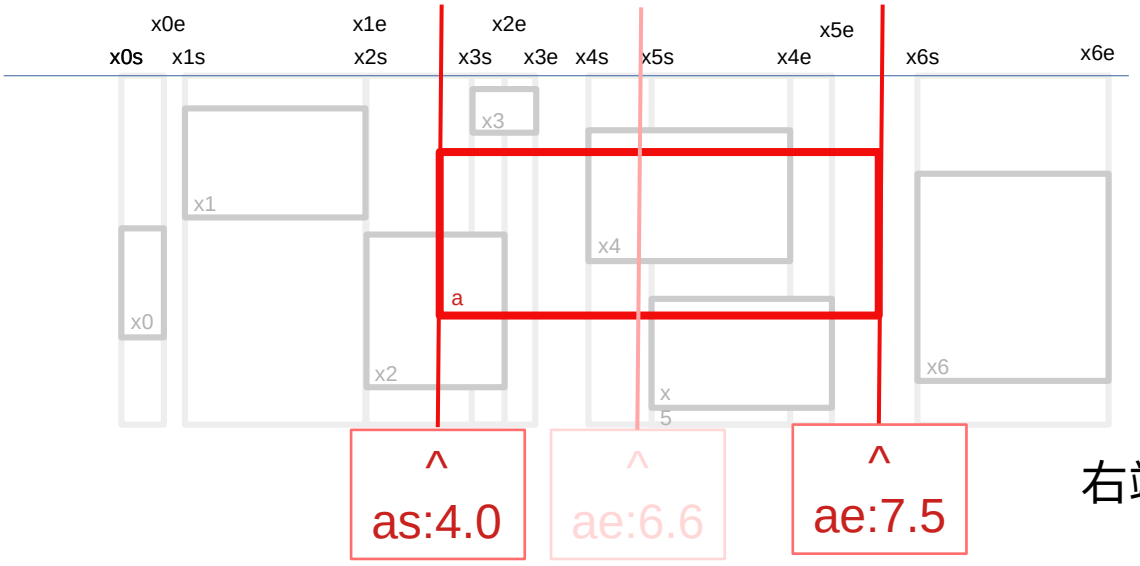
右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

as:4.0

抽出範囲が動いた時

- すると、「a の右端より左側の範囲」と重なっている X の要素が増える（x5 が追加）



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

^ ae:6　^ ae:7.5

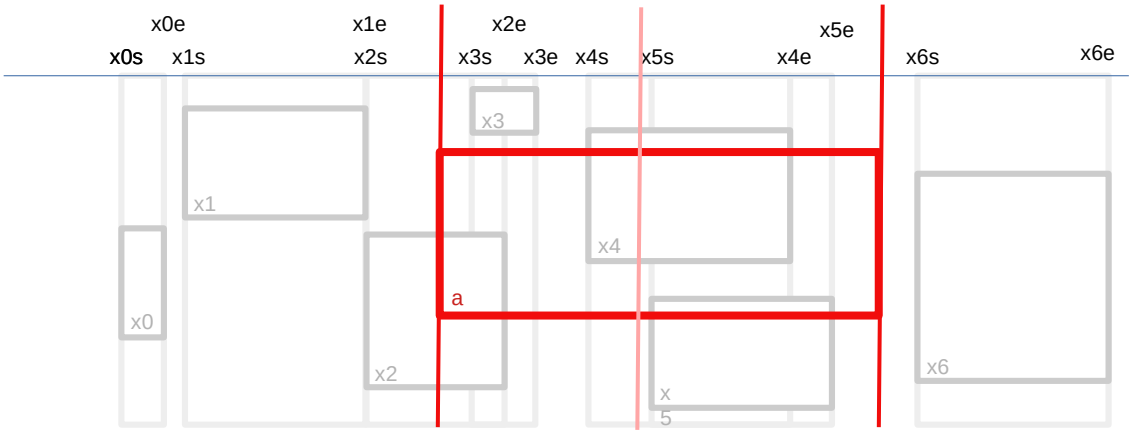
右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

^ as:4.0

抽出範囲が動いた時

- このとき、抽出 X の要素が変化するが、変化したのは x5 のみ
  → 抽出 X(hash_table) は、 x5 を追加するだけで更新ができる!
  (右端テーブルをチェックする必要はない)



左端

x0	x1	x2	x3	x4	x5	x6
0.0	0.8	3.0	4.5	6.2	6.7	8.7

^ ae:6　^ ae:7.5

右端

x0	x1	x2	x3	x4	x5	x6
0.6	3.0	5.0	5.5	8.0	8.3	9.9

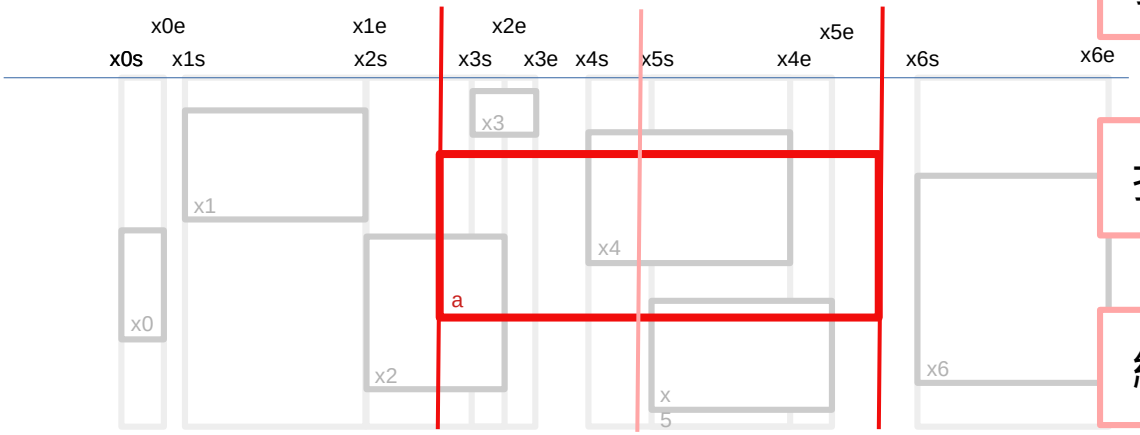
^ as:4.0

抽出 X (hash_table) :

x2	x3	x4	x5

抽出範囲が動いた時

- 抽出 X に追加されたオブジェクト {x5} が、抽出 Y にもあるか確認
- あれば抽出、なければ保持



抽出 X : x2 x3 x4 x5

抽出 Y : x0 x1 x2 x4 x5 x6

結果 : x2 x4 x5

本アルゴリズムによる実装

1) ポインタをずらす

2) ポインタのテーブル上での変化を取得

3) 抽出 X (Y) テーブルに格納

4) 抽出 Y (X) テーブルに要素が存在するかチェック
   あれば抽出結果テーブルに格納、なければ抽出 X (Y) テーブルに保持

   この手順を、 4 ポインタすべてに適用する
   (言葉にしたら複雑な処理に見えるが、実際そうでもない。ただただ説明が難しい)

本アルゴリズムによる実装

- このとき、
  すべてのオブジェクトの数を N
  抽出情報の更新されたオブジェクトの数を M として
  時間計算量は O(M) となる

  →抽出範囲の変化が少なければ少ないほど、計算量が小さくなる
  (変化がない場合は、 4 つのポインタの値を更新するだけなので O(1))
  (逆に、もんのすごく大胆に変化する場合は、最悪 O(N) になってしまう)

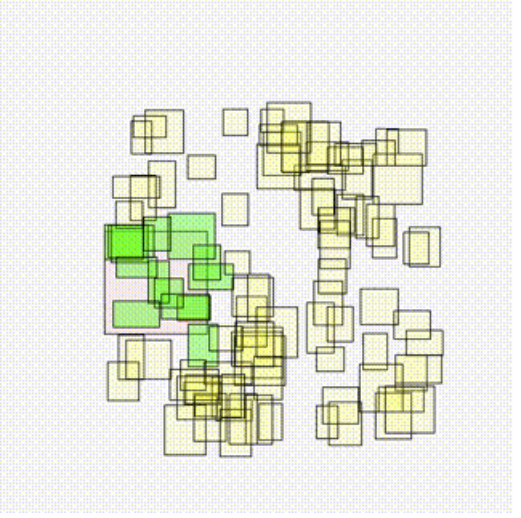
計算量の変化

1 回の更新あたりの移動量が少ない

→ ほぼ O(1) を維持する

1 回の更新あたりの移動量が多い

→ O(N) に近づく



III . 従来手法との比較

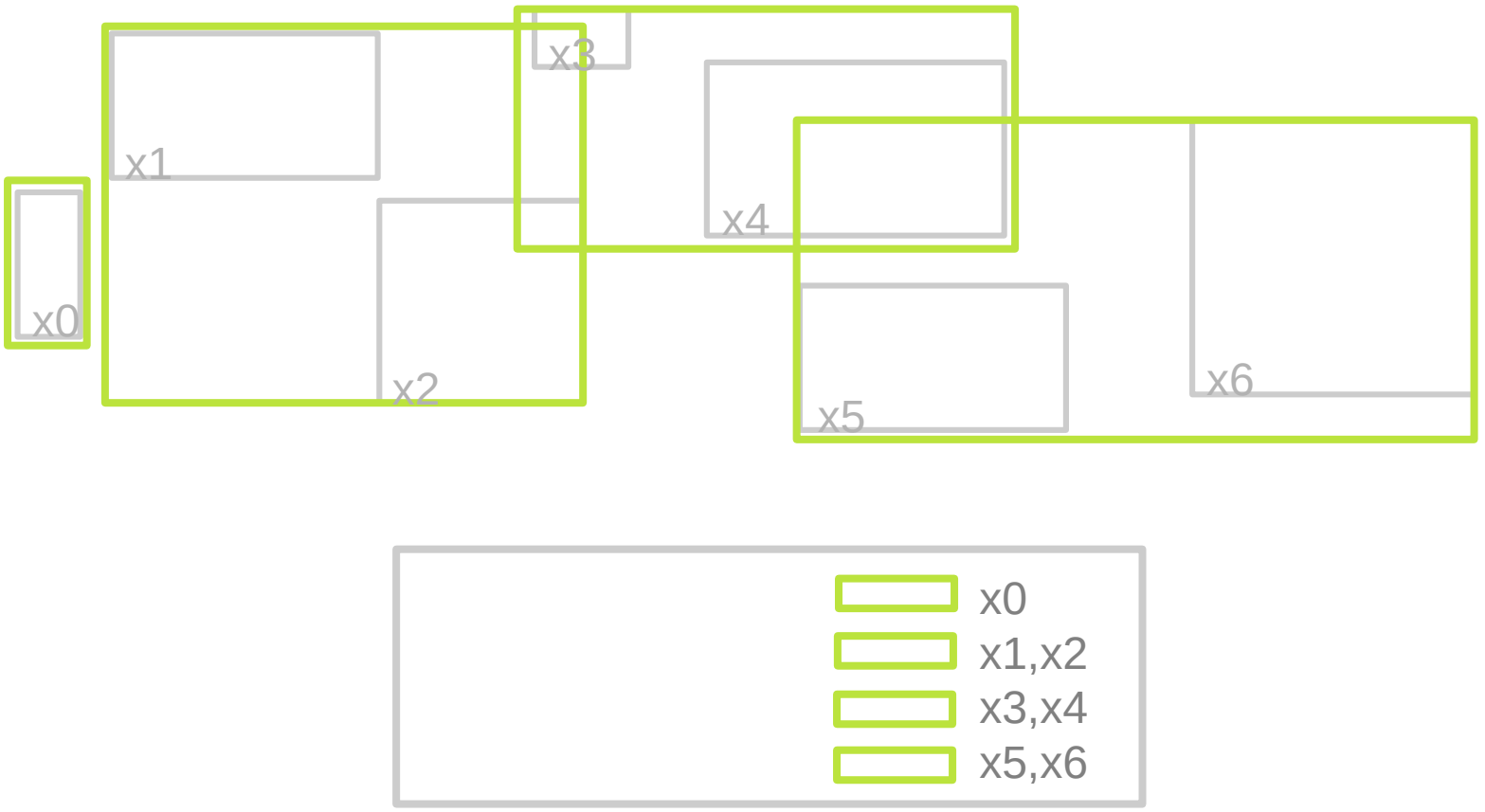
比較対象となるアルゴリズム

- R-tree

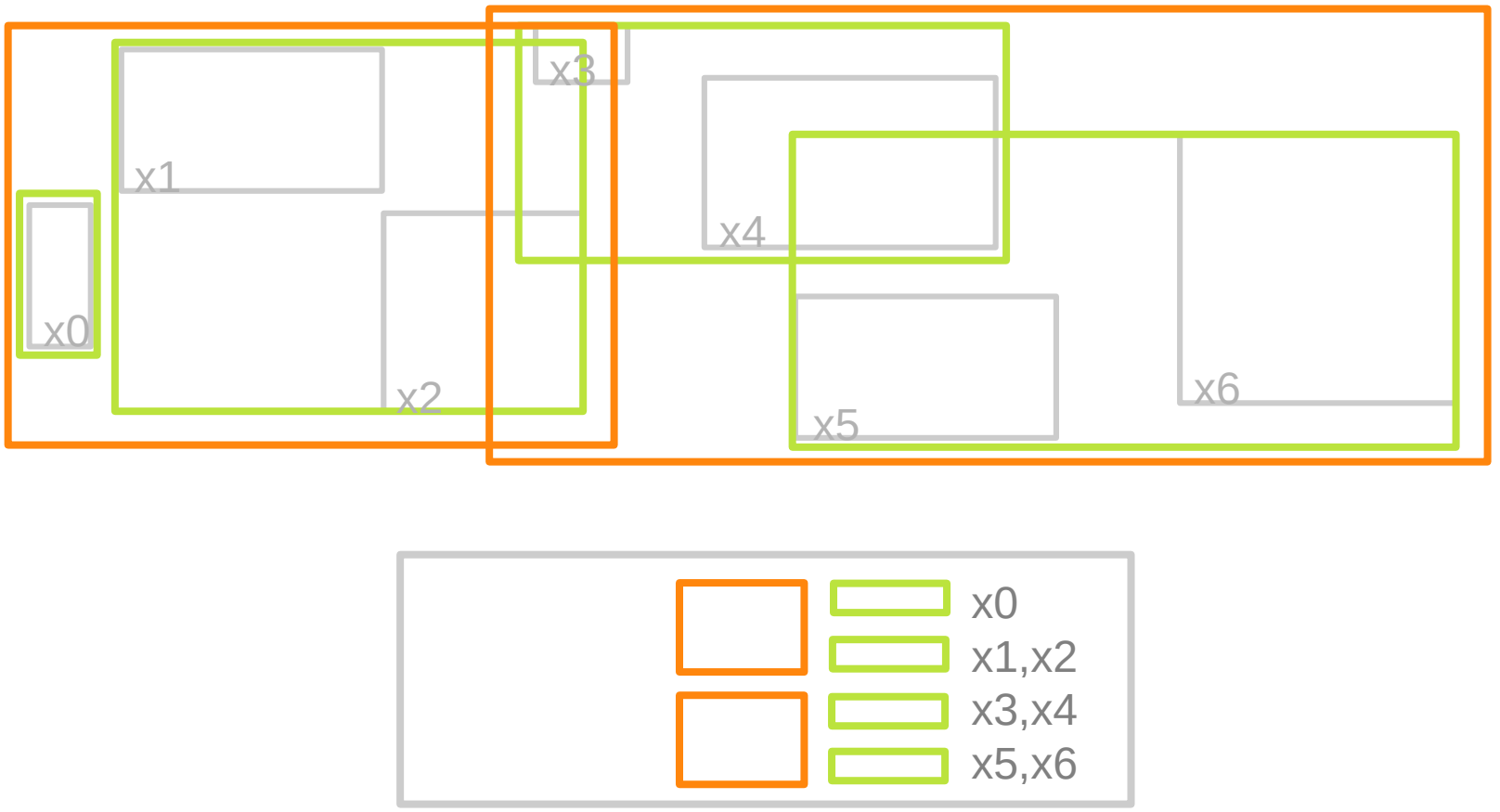
  空間オブジェクトを 2~3 個ずつ集めてそれぞれをグループ化
  そのグループをまた 2~3 個ずつ集めてそれぞれをグループ化
  そのグループをまたまた 2~3 個ずつ集めてグループ化 ... と
  していくことで、全ての空間オブジェクトを階層化して
  うまくまとめ上げる木構造アルゴリズム (次に図示)

  空間インデックスの手法としては
  最もメジャーであり、多くのパターンにおいて最適な計算量を発揮

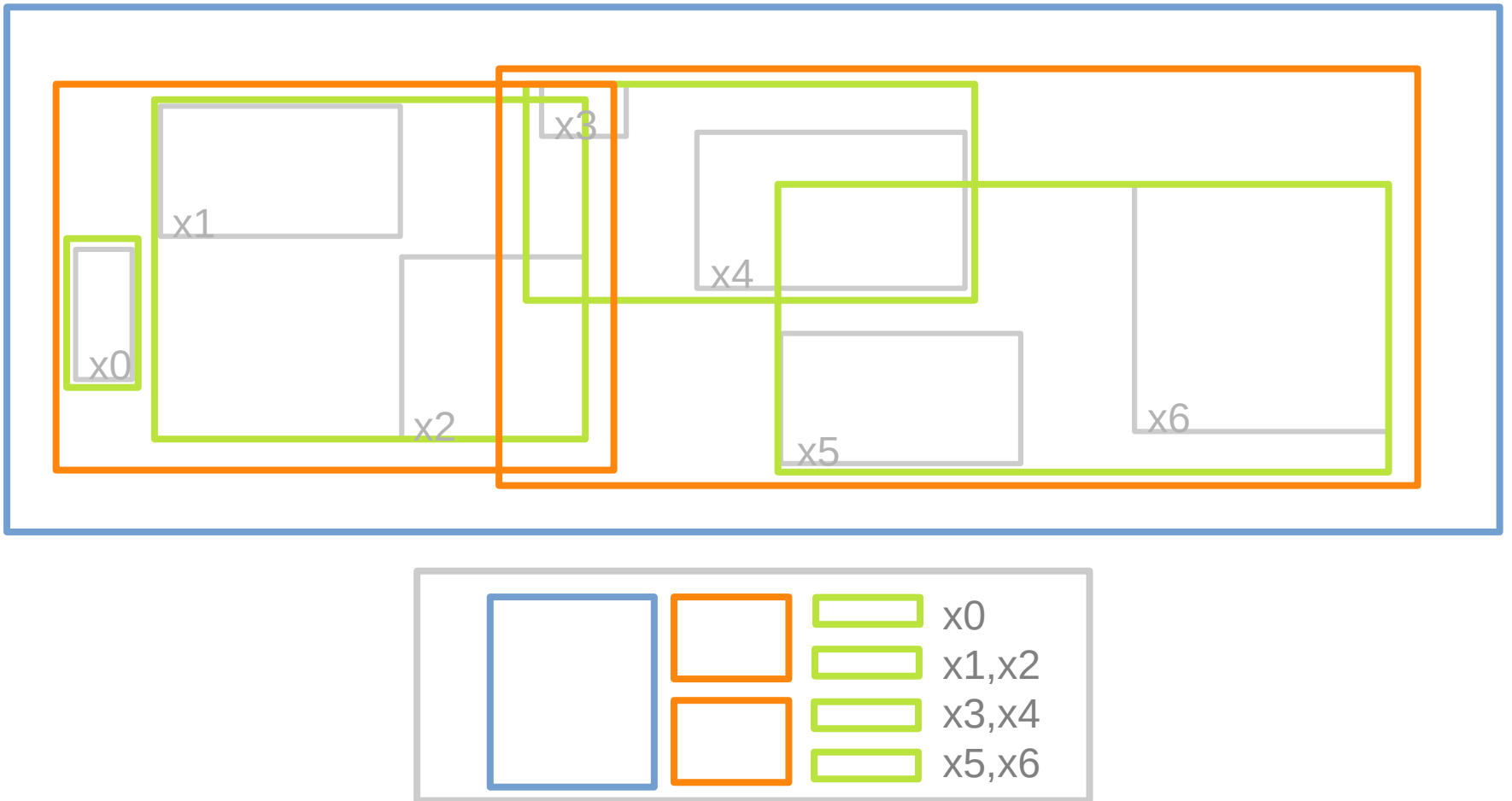
R-tree 例



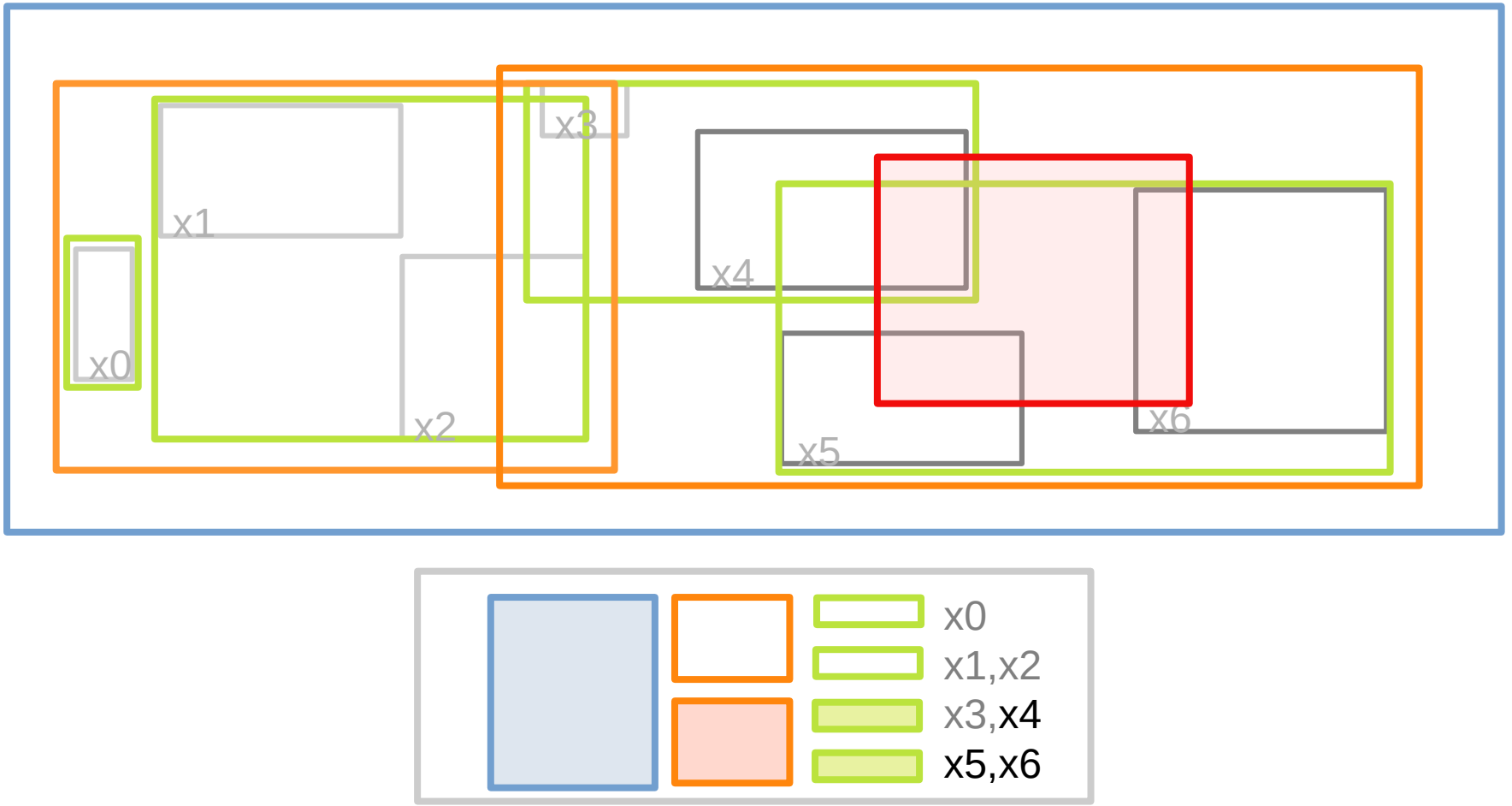
R-tree 例



R-tree 例



R-tree 例



R-tree と本アルゴリズムの比較

Algorithm	R-tree	本アルゴリズム
値の挿入 / 削除	O(logN)	O(logN) [二部探索]
範囲抽出	O(1) ~ **O(logN)** ~ O(N)	O(N)
抽出範囲の更新	-- 範囲抽出を再実行 --	**O(1)** ~ O(N)

※R-tree の場合、 log の底は「各親オブジェクトの持つ子オブジェクトの最大数」となる。したがって 2 とは限らない (ただし 2 や 3 が多い)

抽出範囲に変化が一切ない場合 → R-tree が必ず最適
更新される時 → 多くの場合、本アルゴリズムが最適

Ⅳ. 現状と課題

課題 (1)：活動自体の問題

- 文章化、及び証明ができていない (できない)
  < そもそも (論文のような) 文章の書き方がよくわかっていない！ >

自分で調べるだけでは限界があるので
情報科学の達人プログラムを通じて、 (申し訳ないのですが)
研究者の方からのアドバイスをお願いしていきたい ...
(宜しくおねがいします)
(すでに何人かの方には非常にお世話になっているのですが、
それに関する言及はプレゼンテーションの最後に ...)

課題 (2)： アルゴリズムに関する課題

- 実装がとにかく重い
  現時点の実装だと (時間計算量 O(1) を追求するには)
  hash テーブル (maxsize:N) x 3 + 連想配列 (size:N/4) x 4
  といった地獄みたいなコンビネーションが必要
  (空間計算量も、一応 O(N) だが係数がデカい)

  他のライブラリ (標準ライブラリ含め) が使えない状況では
  実装のハードルがものすごく高い

  今後、コンテナの最適な構造を追求していく必要がありそう

課題 (2)： アルゴリズムに関する課題

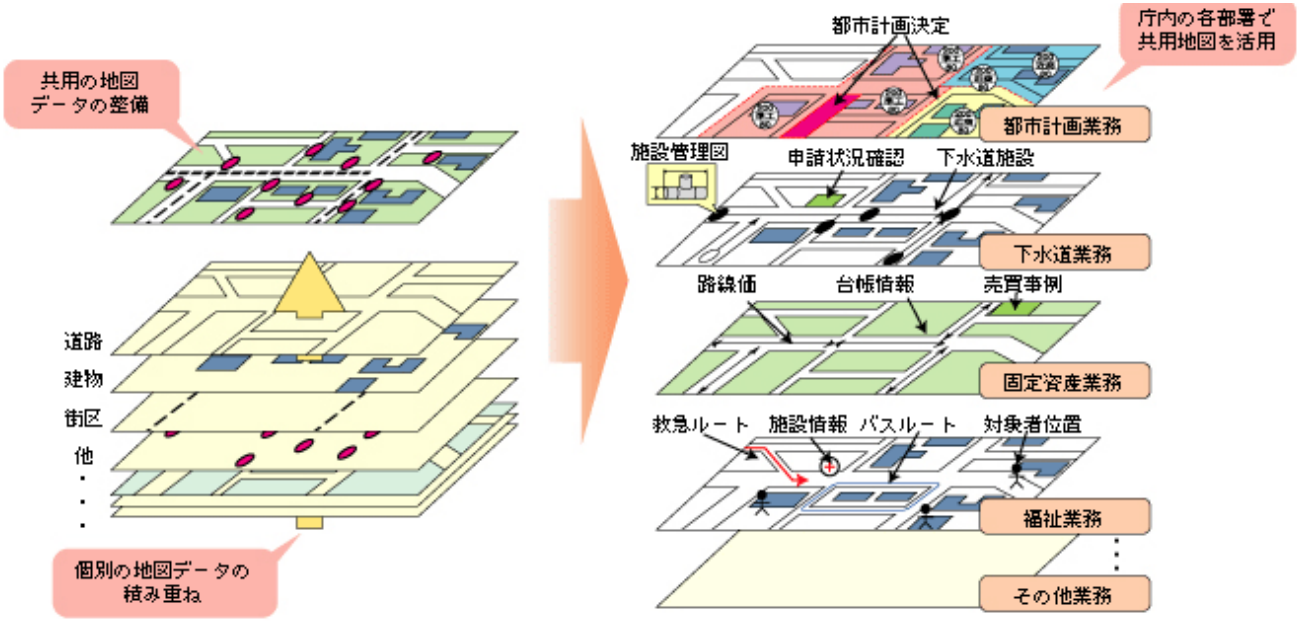
- コンテナへの新規の要素 (空間オブジェクト) の
  挿入 / 削除 を行う機能の実装が未だうまくいっていない

... がんばる… (発表までに間に合わず)

Ⅴ. その他 / まとめ

こんな用途に使えそう (1)

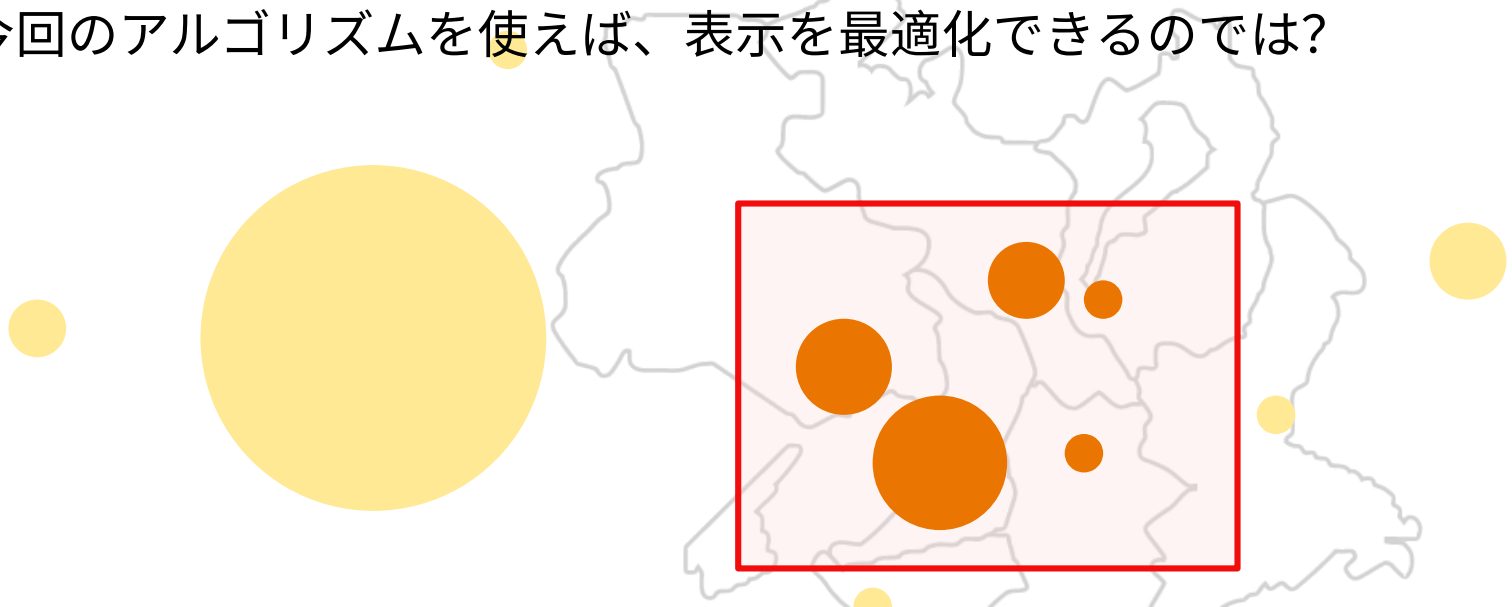
- 地理情報システム (GIS)
  地理的位置を手がかりに、位置に関する情報を持ったデータ（空間データ）を
  総合的に管理・加工し、視覚的に表示する技術



出典：総務省
平成 16 年版 情報通信白書
https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h16/html/G3504400.html

こんな用途に使えそう (1)

- 地理情報システム (GIS)
  地理的な位置に対応した情報
  (地名や、「人口」などの統計データなどいろいろ ...)
  のそれぞれの表示範囲を、点 or 矩形として捉えた上で、
  今回のアルゴリズムを使えば、表示を最適化できるのでは?

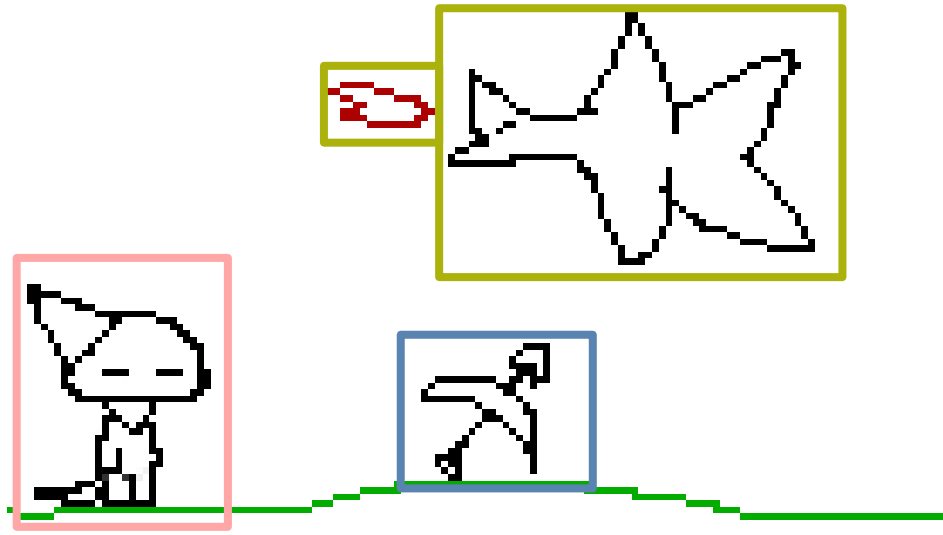


こんな用途に使えそう (2)

- ゲームプログラミング
  主に、当たり判定の最適化

例えば 2D アクションなら →
プレイヤーの MBR を抽出範囲に、
敵、アイテムの MBR を
抽出対象オブジェクトとすれば、
「プレイヤーとぶつかっている」
オブジェクトのみを、フレームあたり
定数時間に近い時間で
取得することができる

MBR (最小外接矩形) とは?
簡単に言うと、一つのオブジェクトを
完全に囲むことのできる最小の矩形



こんな用途に使えそう (2)

- ゲームプログラミング
  主に、当たり判定の最適化

  この場合、動く空間オブジェクトには考慮が必要
  (本アルゴリズムでは、動く空間オブジェクトの抽出を
  局所的に行うことができないため <R-tree も同様だが > 、
  該当オブジェクトのみ導入を諦めるか特別な処理が必要)
  例えば、動く範囲が一定範囲内に収まっているオブジェクトであれば、
  その範囲の MBR をアルゴリズムに適用すれば、最適にはならなくともなんとか計算量を抑えることができる

最後に …

- 本研究にはまだ解決すべき課題や伸びしろがいくつもあるので
  さらなる改善・発展に向けてしっかりと続けていきたい

今回の自主研究及び発表について、
様々な形でご協力いただいた Ｙ教授（本スライドでは伏せ字）
そして、今回の発表の機会を与えてくださった
情報科学の達人のメンターの皆様、本当に感謝しています

ご清聴ありがとうございました