DP特集

菅原研M2　浜田　大

DP

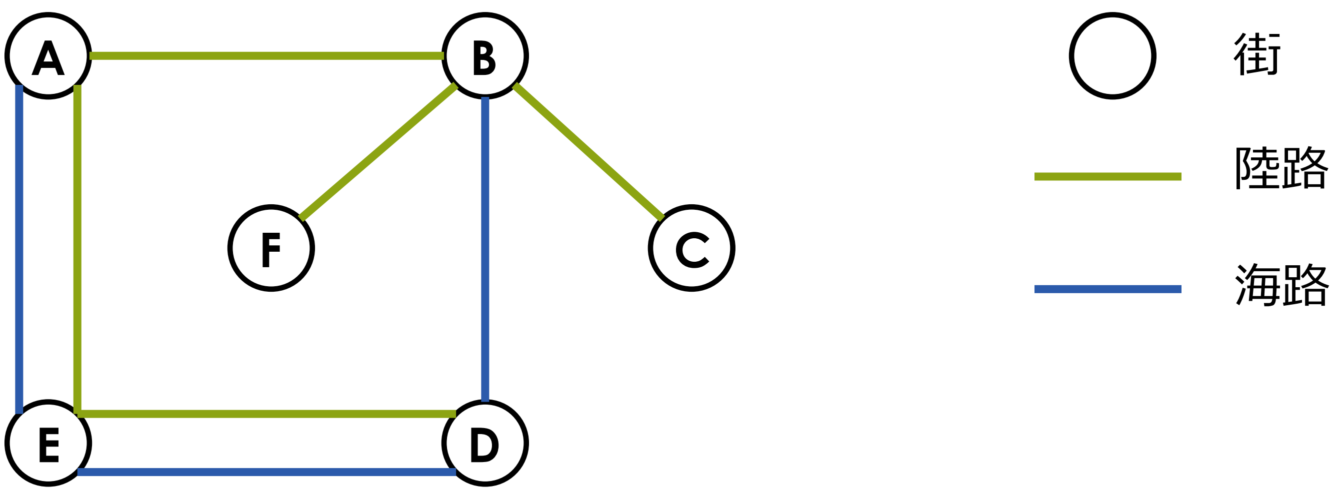
- Dynamic Programming(動的計画法)　略してDP
- DPで解ける典型的な問題を知っておく。実際に書いてみる
  - ナップザック問題
  - 最長増加部分列
  - 文字列の編集距離 (情報系の生物学でおなじみ)
  - 巡回セールスマン問題 (bitDPによる)

今回扱う問題

- 2010年模擬国内予選/D: Mr.Rito Post Office
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2200

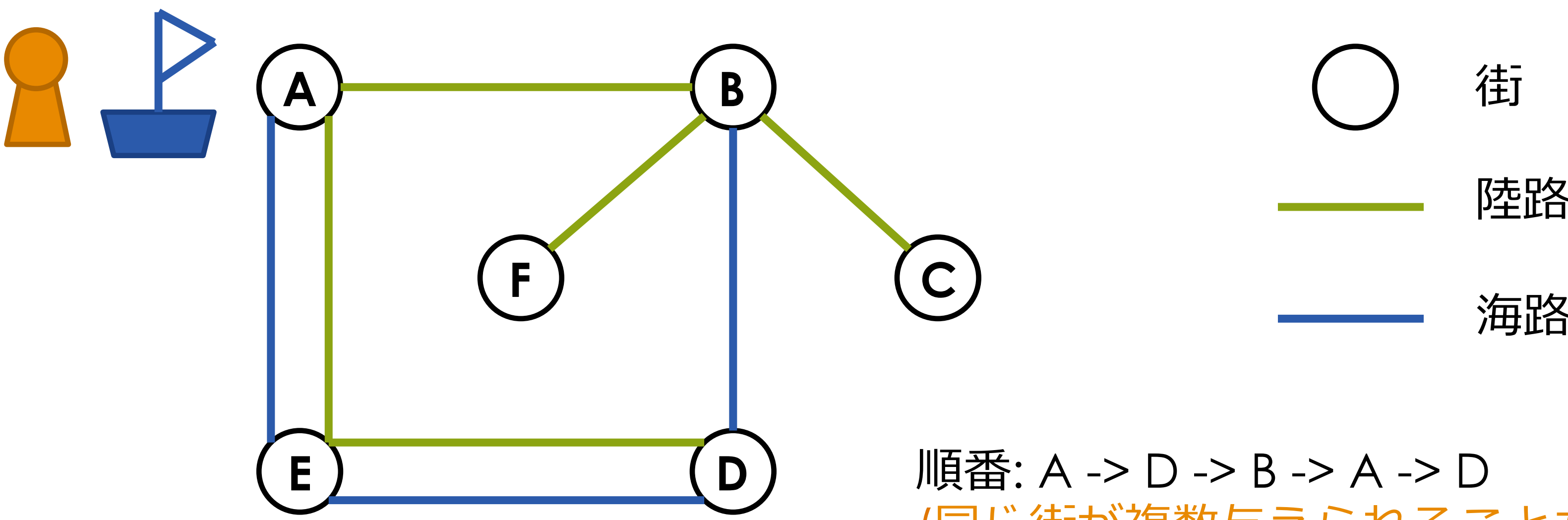
概要

- 街とそれらを結ぶ道（陸路、海路がある）から成るグラフが与えられる



概要

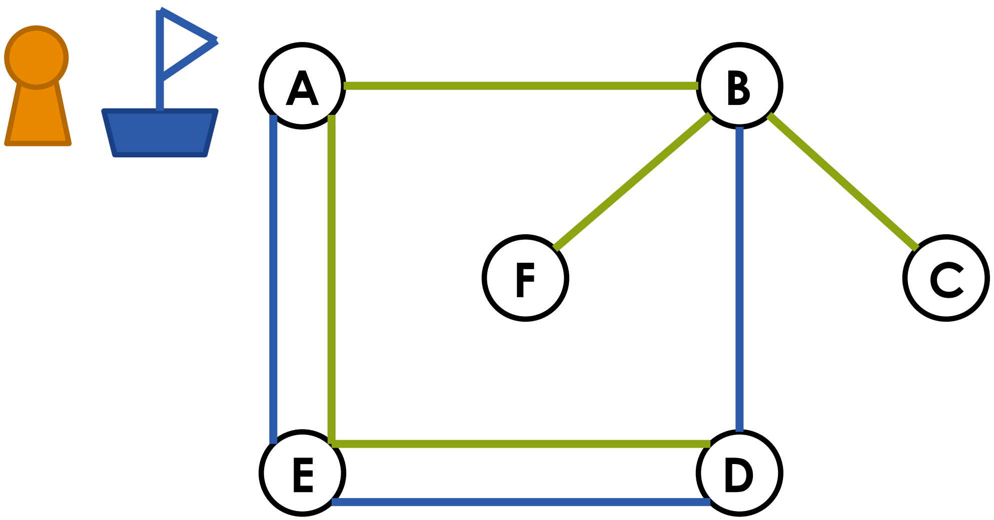
- 利藤さんは船とともにA(z_1)にいる
- 街の列が与えられるので、その順番で効率的に訪れたい。



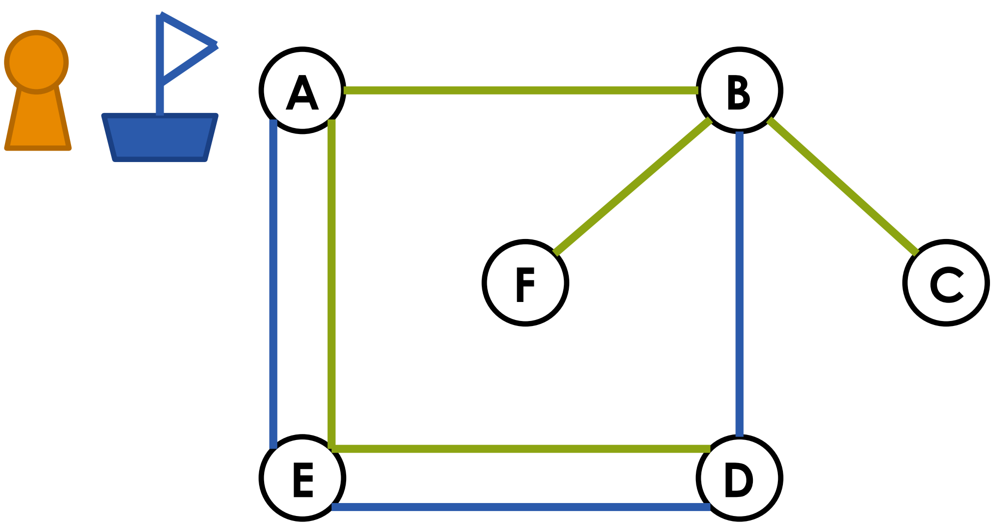
順番: A -> D -> B -> A -> D
(同じ街が複数与えられることもある)

問題を考える時の鉄則

- 簡単な例から考えよう！
  - 例えば、A -> D までの探索はどうすればいい？

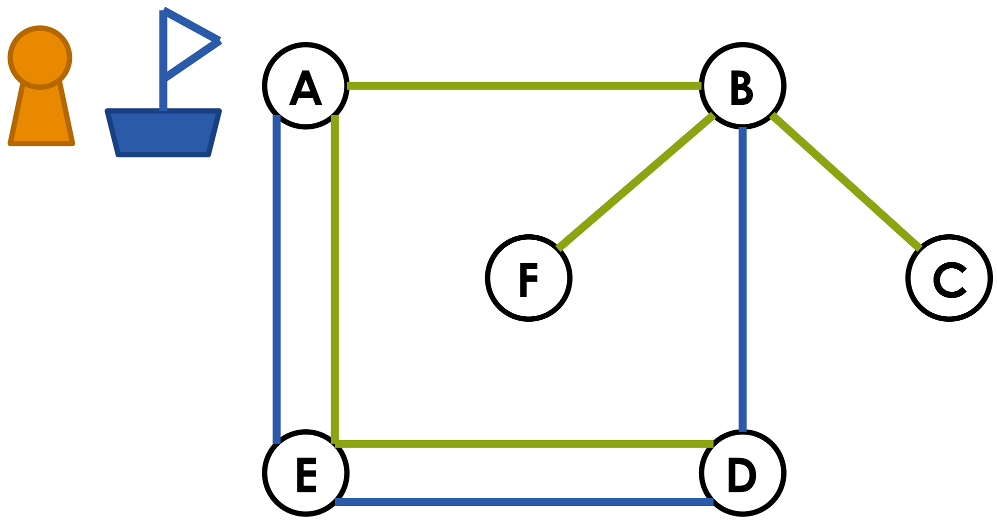


列挙



- A ▶ E ▶ D
  - 20
- A ▶ E ▶ D
  - 15
- A ▶ E ▶ D
  - 10

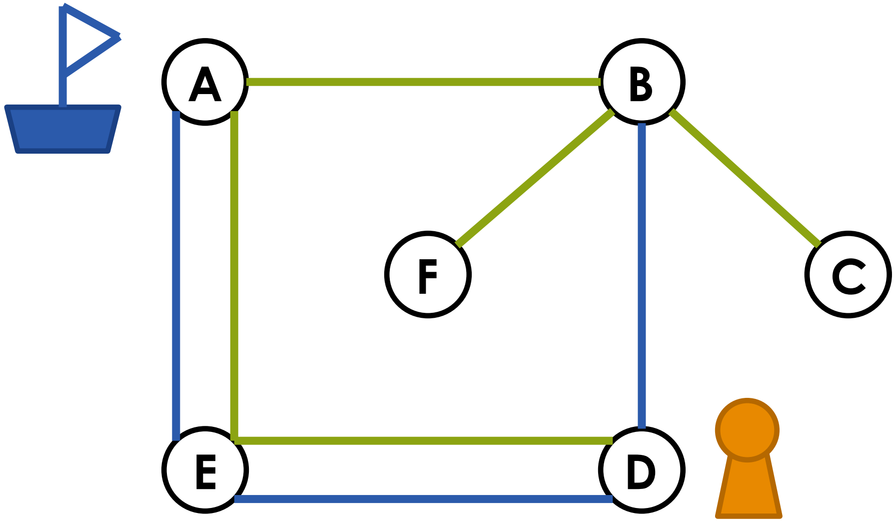
一番いいのを頼む



- A ▶ E ▶ D
  - 20
- A ▶ E ▶ D
  - 15
- A ▶ E ▶ D
  - 10

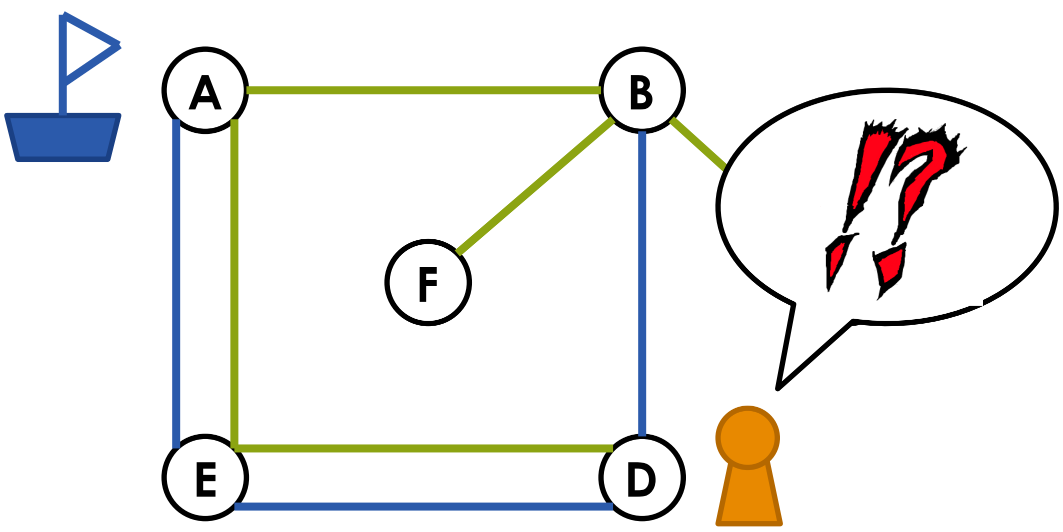
これがベストかな・・・

Dに陸路で来てみた



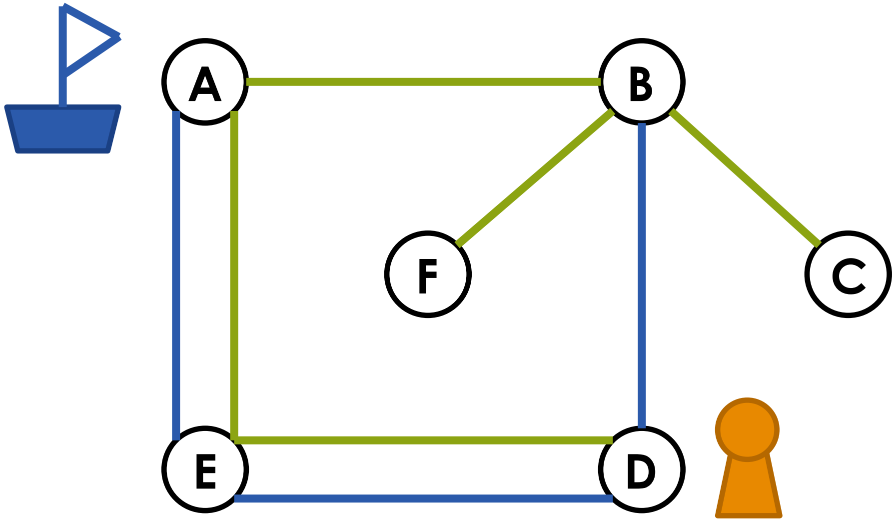
ところが

- 次は、Bに行ってねと言われました



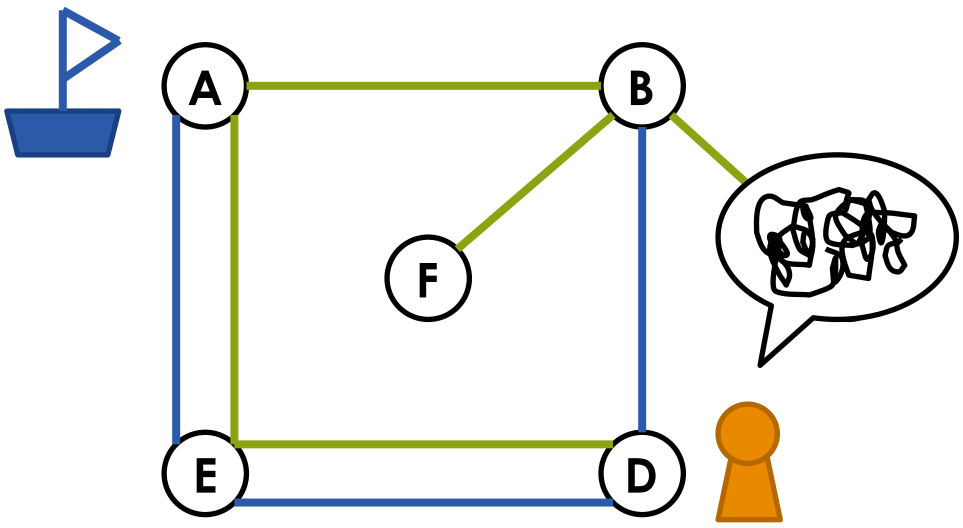
利藤さんの反省

- 次は、Bに行ってねと言われました
  - この状態でBに行くには、一度Aまで戻る必要がある
  - 一方、海路でDまで来ていれば D ▶ B でよかった



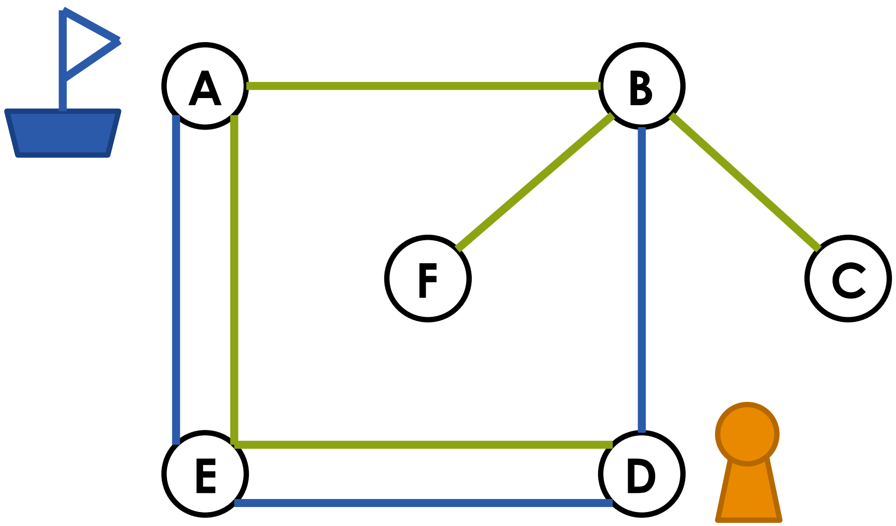
貪欲法ではうまくいかない

- 集荷の移動の度に、それぞれで最適な戦略(最も時間がかからない方法)をとっていく、という方針ではダメそうだ。
  - まじめに探索をする（複数の状態を保持する）必要がありそう。



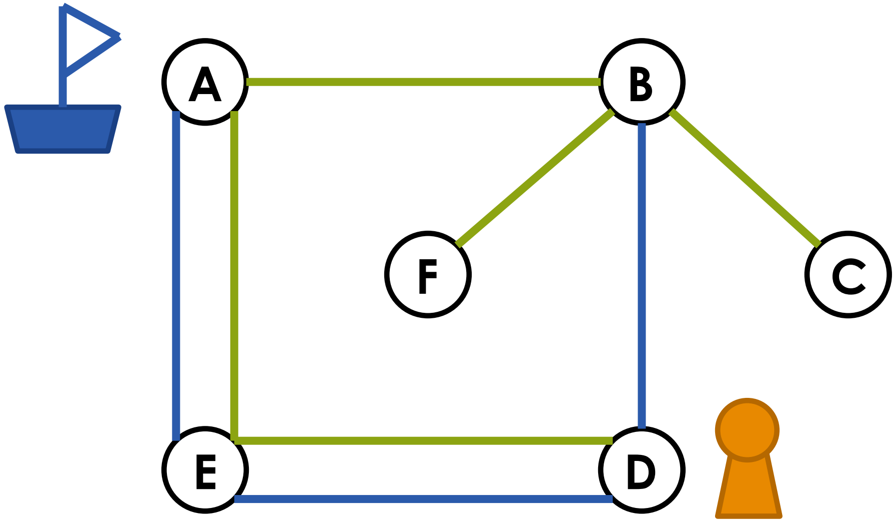
状態を増やそう！

- 大事なのは、自分がDにいるとき「船がどこにあるか」
  - Aにある場合、Eにある場合、Dにある場合 の 3通り



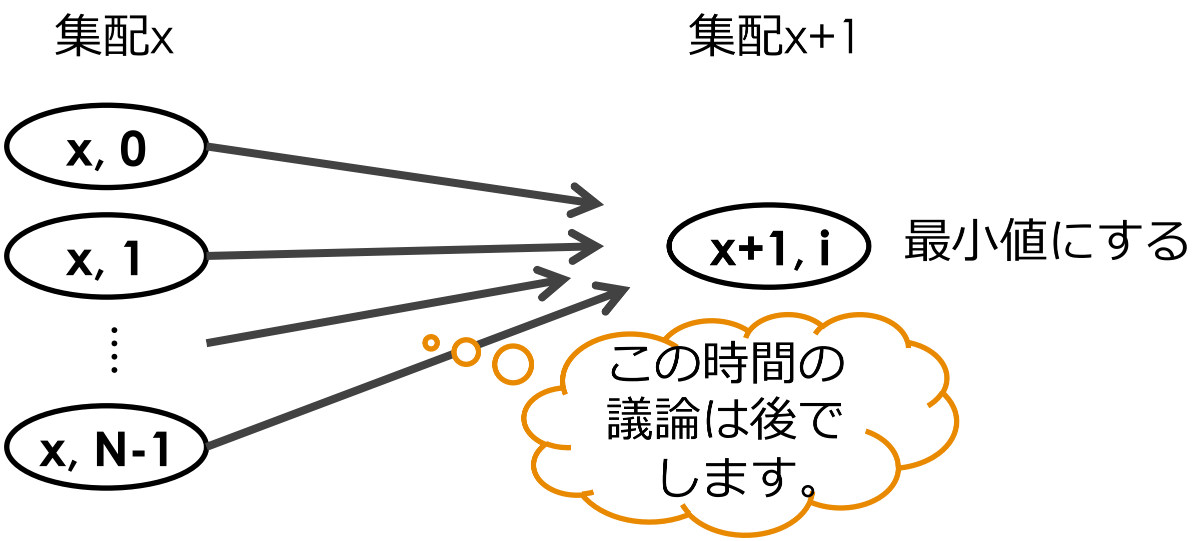
状態を増やそう！

- それぞれの状態から、次にBに行くまでの最小経路を求め、更新してやればいい



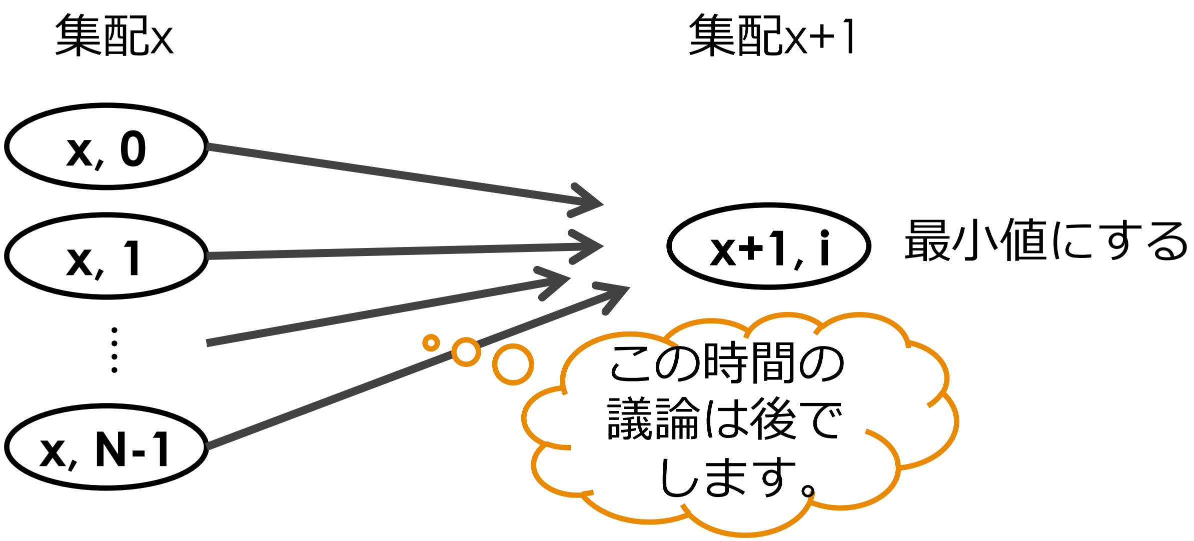
より一般的に言うと

- 現在の状態を {何番目の集配が終わった, 船がどこにあるか} ={x, s} と置く。状態には最短時間を格納しておく
- {x+1, i} を更新したい時は次のようにする。



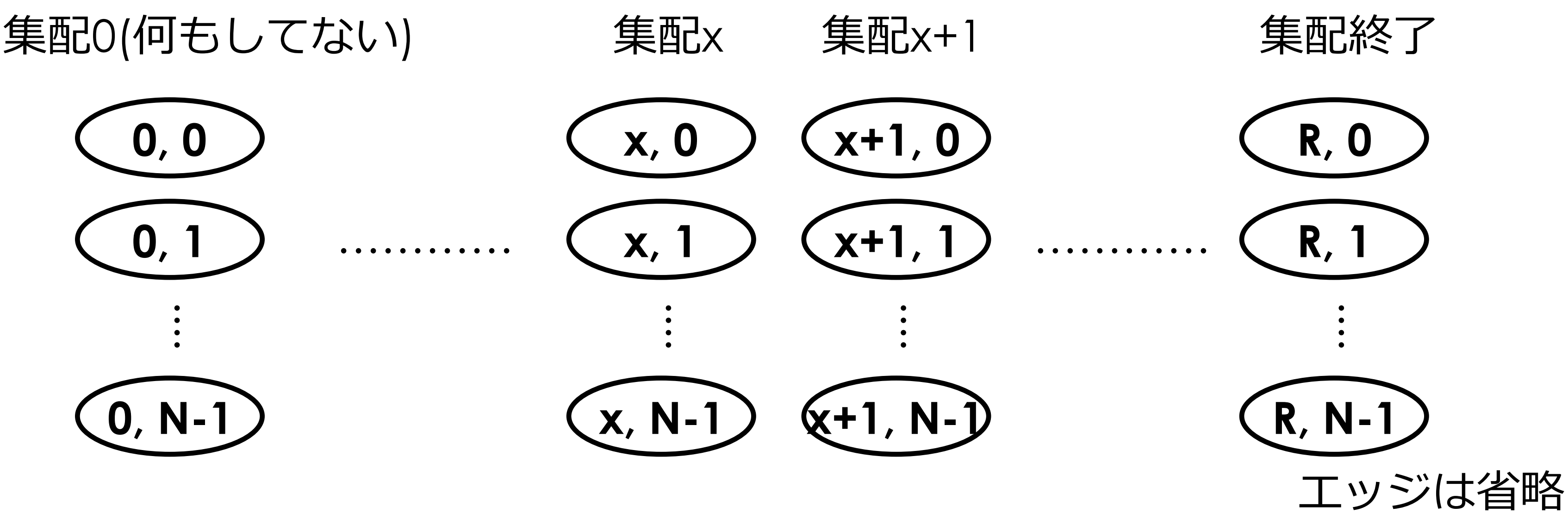
より一般的に言うと

- {x+1, 0〜N-1} について、同じ事を繰り返します。
  - これにより、前の状態から順番に決めていくことが可能です。



イメージしろ

- 頂点が 集配の数 * 街の数 だけ並んだグラフの姿を！
  - 何も集配してない状態から、終了状態まで順番に求められます。



実装はどうなるの

- 通常、配列を使って表します。
  - dp[x] [i] := 集配がxまで終わり、船がiにあるときの時間の最小値
- 配列サイズ
  - 集配の数は 1,000
  - 街の数は 200
    - なので、配列サイズは 1,000 * 200 = 200,000
    - これが 100,000,000 ぐらいになるとちょっと危険
      - 状態を増やしすぎてませんか？

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);

dp[1][go[0]] = 0;
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);
```

初期化
INF: 十分大きな値

```
dp[1][go[0]] = 0;
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);

dp[1][go[0]] = 0;
for (int i = 0 ; i < R ; i++) {
  for (int
    if (dp
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

初期状態を与える
初期位置：集配0 なので、
はじめの集配は終わっていると考える。

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fi
```

次の集配 i で、船が j に停泊しているとき
状態にたどり着いてない (=INF) ならスキップ

```
dp[1][go[0]] = 0;
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                        dp[i][j] + (何か)
    }
  }
}
```

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);

dp[1][go[0]] = 0;
for (int i
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

次の集配先に、
船を k に停めた状態でたどり着くことを考える

更新式の実装例

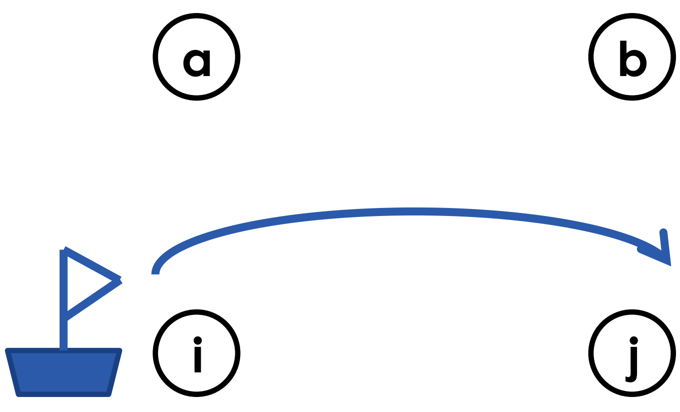
```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);

dp[1][go[0]] = 0;
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

？？？

何かをどうにかしよう

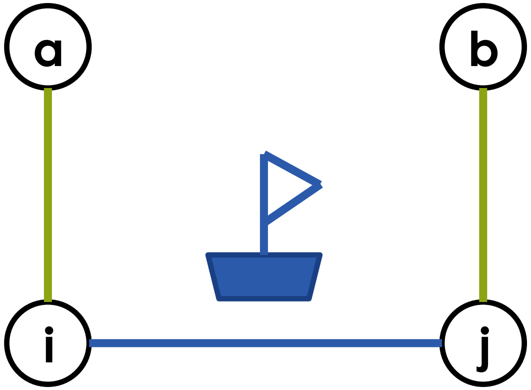
- 今、船が i にあって、a から出発し
  船を j においた状態で b にたどり着くには？



気付こう

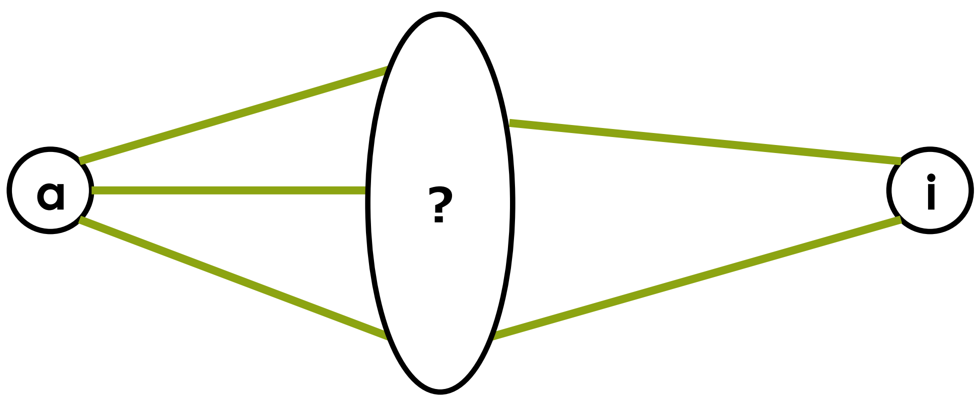
- a -> i は陸路
- i -> j は海路
- j -> b は陸路

- で行く必要がある。a -> b まではそれぞれの最短距離の和。



最短路

- a -> i を陸路だけで行く事を考える。
  - dfs (端点同士の最短路)
  - ダイクストラ (単一始点最短路)
- どちらでも求まるが、思い出して欲しい



更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i = 0 ; i < R+1 ; i++)
  Arrays.fill(dp[i], INF);

dp[0][0] = 0;
for (int i = 0 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] >= INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か))
    }
  }
}
```

更新式の実装例

```
int[][] dp = new int[R+1][N];
for (int i
  Arrays.f

dp[0][0] = 0;
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
                       dp[i][j] + (何か)
    }
  }
}
```

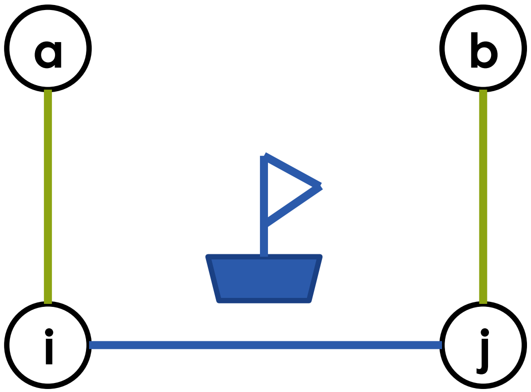
最大で R * N * N = 40,000,00回かかる！
ダイクストラなんてしてられない！
おまけにダイクストラは実装が重い！

もっと良い道具を僕らは知っている

- 街の数が少ない (N <= 200) ので
  - 全点対最短路を効率良く求める方法がある！
- ご存知、ワーシャルフロイド法
  - DPする前に、陸路、海路それぞれについて、最短路を求め
    - landpath[a][b]
    - seapath[a][b]
  - なんかに格納しておく
  - 計算量は 200 * 200 * 200 = 8,000,000

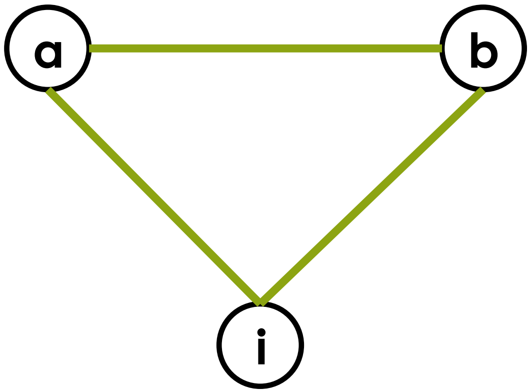
何かをどうにかしよう

- よって、「何か」の部分は、
  - landpath[a][i] + seapath[i][j] + landpath[j][b]
- で書ける。



もう一つ考えるべきケース

- 船を移動しない時 (j = i)
  - landpath[a][i] + landpath[i][b] より landpath[a][b] の方が速い場合がある



更新式の実装例

```
for (int i = 1 ; i < R ; i++) {
  for (int j = 0 ; j < N ; j++) {
    if (dp[i][j] == INF) conitnue;
    for (int k = 0 ; k < N ; k++) {
      dp[i+1][k] = min(dp[i+1][k],
         dp[i][j] + landpath[from][j]
 + seapath[j][k] + landpath[k][to]);
    }
    dp[i+1][j] = min(dp[i+1][j],
         dp[i][j] + landpath[from][to]);
  }
}
```

陸路のみで移動するパターンを追加

まとめ

- 陸路、海路だけを使う場合の最短路を
  ワーシャルフロイド法で求める
- {集配の処理数, 船がある街} を状態に持って更新DP
- 答えは {R, 0〜N-1} の最小値

- 簡単じゃなかった・・・(´・ω・｀)
  - ぱっと見簡単だとは思った

細かい実装上の注意

- 問題文の条件
  - 陸路または海路が2本以上存在することがある
    - ArrayList などで保持する方法もあるが、
      WFがやりにくいのでコストが大きい方は無視していい
  - 初期状態では利藤さんと船はともに港町 z_1 に存在する
    - 確認しよう

細かい実装上の注意

- dp[a][b] = INF
  - 状態 {a, b} へ到達できない
  - INFを大きくしすぎるとDP計算で
    オーバーフローすることがある
  - 街の数が 200, 移動時間が 1,000 なので
    - 一回の集配に最大 200,000
    - 最大 1,000 回の集配に 200,000,000
  - int だとあと数十倍するとオーバーフロー
- この場合素直に long を使った方がいい。

お疲れ様でした。

http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=439492