KHOA CÔNG NGHỆ THÔNG TIN

**BỘ MÔN KỸ THUẬT PHẦN MỀM**



**BÁO CÁO**

**HỌC PHẦN: TRÍ TUỆ NHÂN TẠO**

**ĐỀ TÀI: CÀI ĐẶT MỘT SỐ THUẬT TOÁN AI**

**Giáo viên giảng dạy: Nguyễn Đình Cường**

Họ và tên: Nguyễn Thành Lãnh

MSSV: 61133851

Lớp: HETTNT

Nha Trang - năm 2022

**DANH SÁCH THUẬT TOÁN**

[I. Lập trình trò chơi Puzzle 8-puzzle 1](#_Toc111404242)

[1.1 Lý thuyết 1](#_Toc111404243)

[1.2 Thuật toán. 2](#_Toc111404244)

[1.3 Cài đặt 3](#_Toc111404245)

[II. Giải bài toán tam giác bằng biểu diễn tri thức mạng ngữ nghĩa 5](#_Toc111404246)

[1.1 Lý thuyết 5](#_Toc111404247)

[1.2 Giải bài toán tam giác bằng biểu diễn tri thức mạng ngữ nghĩa: 6](#_Toc111404248)

[1.3 Thuật toán 8](#_Toc111404249)

[1.4 Cài đặt 9](#_Toc111404250)

[III. Giải thuật Quinlan 11](#_Toc111404251)

[1.1 Lý thuyết 11](#_Toc111404252)

[1.2 Thuật toán 12](#_Toc111404253)

[1.3 Cài đặt thuật toán. 12](#_Toc111404254)

[IV. Giải thuật Robinson 14](#_Toc111404255)

[1.1 Lí thuyết 15](#_Toc111404256)

[1.2 Thuật toán 16](#_Toc111404257)

[1.3 Cài đặt thuật toán 17](#_Toc111404258)

[V. Giải thuật vương hạo 21](#_Toc111404259)

[1.1 Lí thuyết 21](#_Toc111404260)

[1.2 Thuật toán 21](#_Toc111404261)

[1.3 Cài đặt thuật toán 22](#_Toc111404262)

[VI. Thuật toán di truyền. 29](#_Toc111404263)

[1.1 Lí thuyết 29](#_Toc111404264)

[1.2 Thuật toán 30](#_Toc111404265)

[1.3 Cài đặt thuật toán. 31](#_Toc111404266)

[VII.Thuật toán suy diễn tiến 31](#_Toc111404267)

[1.1 Lí thuyết 31](#_Toc111404268)

[1.2 Thuật toán 31](#_Toc111404269)

[1.3 Cài đặt thuât toán 31](#_Toc111404270)

[VIII. Cài đặt thuật toán duyệt chiều rộng (BFS) 31](#_Toc111404271)

[1.1 Lí thuyết 32](#_Toc111404272)

[1.2 Thuật toán 32](#_Toc111404273)

[1.3 Cài đăt thuật toán 33](#_Toc111404274)

[IX. Cài đặt thuật toán duyệt chiều sâu (DFS) 34](#_Toc111404275)

[1.1 Lí thuyết 34](#_Toc111404276)

[1.2 Thuật toán 35](#_Toc111404278)

[1.3 Cài đặt thuật toán 35](#_Toc111404279)

[X. Tìm kiếm Heuristic bằng tìm kiếm ưu tiên tối ưu 37](#_Toc111404280)

[1.1 Lí thuyết 37](#_Toc111404281)

[1.2 Thuật toán 37](#_Toc111404282)

[1.3 Cài đặt thuật toán 38](#_Toc111404283)

1. Lập trình trò chơi Puzzle 8-puzzle
2. Lý thuyết

Bài toán 8-puzzle là một câu đố do Noyes Palmer Chapman phát minh và phổ biến vào những năm 1870. Nó được chơi trên lưới 3 x 3 với 8 khối vuông có nhãn từ 1 đến 8 và một hình vuông trống. Mục tiêu của bạn là sắp xếp lại các khối để chúng có thứ tự đúng với mục tiêu đề ra. Bạn được phép trượt các khối theo chiều ngang hoặc chiều dọc vào ô trống.

Có 2 hàm Heuristic:

* Heuristic 1: Tổng số miếng sai vị trí
* Heuristic 2: Tổng khoảng cách sai vị trí của từng miếng.

Việc chọn lựa hàm Heuristic là khó khăn và có ý nghĩa quyết định đối với tốc độ của giải thuật.

Xét lại hoạt động của giải thuật Best First Search:

* Khi có 2 nút cùng có giá trị kỳ vọng đạt đến mục tiêu bằng nhau thì nút có path từ nút bắt đầu đến nút đó ngắn hơn sẽ được chọn trước như vậy nút này có giá trị Heuristic tốt hơn.
* Hay nói cách khác hàm lượng giá Heuristic cho nút gần start hơn là tốt hơn nếu kỳ vọng đến goal là bằng nhau.
* Vậy chọn nút nào nếu kỳ vọng của 2 nút khác nhau? Nút kỳ vọng tốt hơn nhưng xa start hay nút kỳ vọng xấu hơn nhưng gần root

Hàm lượng giá bao gồm cả 2 và có cấu trúc:

**F(n) := G(n) + H(n)**

G(n): phí tổn thực từ root đến n

H(n): phí tổn ước luợng heuristic từ n đến goal.

* Trong bài toán 8 puzzle:

Hàm lượng giá:

**F(n) := G(n) + H(n)**

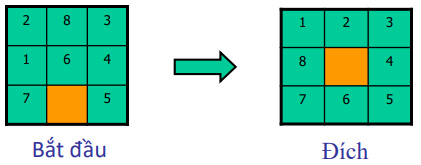
Với G(n): số lần chuyển vị trí đã thực hiện

H(n): Số miếng nằm sai vị trí

Nút X có giá trị heuristic tốt hơn nút Y nếu F(x) < F(y).

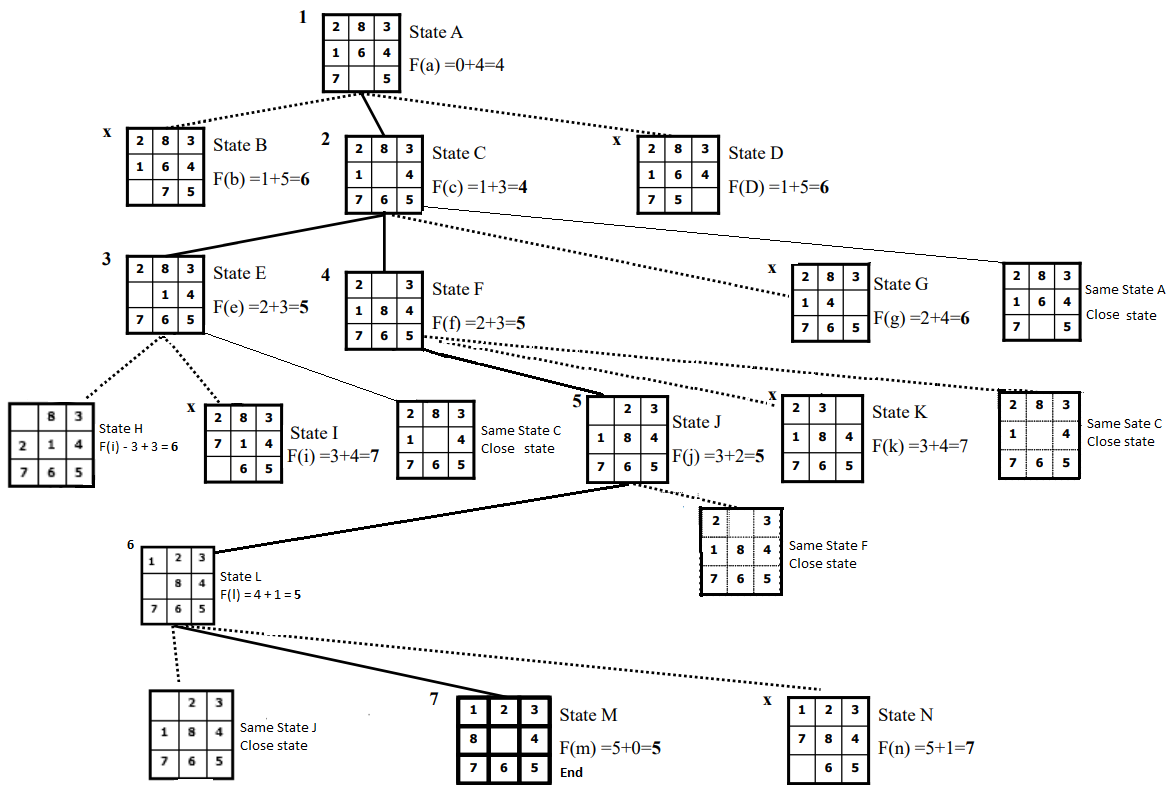
1. Thuật toán.

Với bài toán sau:

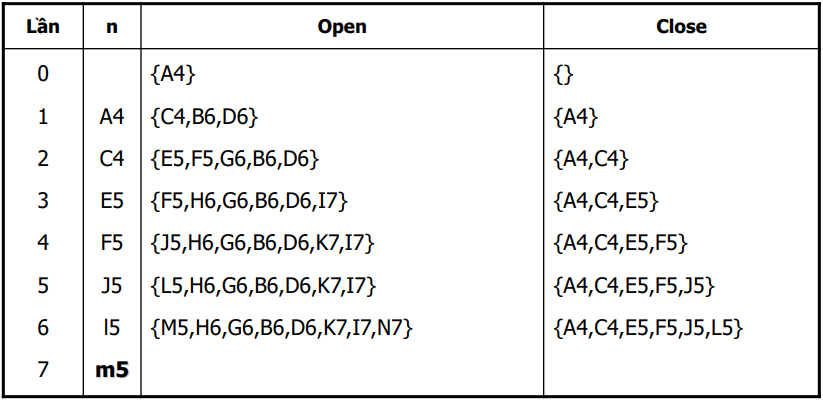


Ta có hoạt động của giải thuật Best First search như sau:

* Bước 1: Đặt OPEN chứa trạng thái khởi đầu.
* Bước 2: Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong OPEN, thực hiện:
* Chọn trạng thái (Tmax) có giá trị H nhỏ nhất trong OPEN (và xóa Tmax khỏi OPEN).
* Nếu Tmax là trạng thái kết thúc thì thoát.
* Ngược lại, tạo ra các trạng thái kế tiếp Tk có thể có từ trạng thái Tmax và kiểm tra.
* Bước 3: Tk đã được xét trước đó hay chưa?
* Nếu chưa thực hiện: f(Tk) = G(Tk) + H(Tk); thêm Tk vào OPEN.

****

Các bước thực hiện:

****

1. Cài đặt

Sử dụng ngôn ngữ lập trình Javascript

* Kiểm tra status bắt đầu có thể đi tới trạng thái cuối hay không

function cantSolution(status) {

  let d = 0;

  for (let i = 0; i < status.length; i++) {

    let c = 0;

    for (let j = i + 1; j < status.length; j++) {

      if (status[j] !== 0 && status[j] < status[i]) c++;

    }

    d += c;

  }

  return !(d % 2);

}

* Đếm số lượng vị trí sai khác giữa hai status

function diff(status1, status2) {

  let d = 0;

  status1.forEach((item, index) => {

    if (item !== status2[index]) d++;

  });

  return d;

}

* Tìm lời giải cho bài toán

async function puzzleAI() {

  const start = [...startStatus];

  const end = [...endStatus];

  const traces = [];

  const statuses = [ [...start] ];

  const actives = [{ status: [...start], step: 0, name: 0 }];

  let step = 0;

  let out = false;

  while (!out) {

    await new Promise((resolve) => setTimeout(resolve, 0));

    step++;

    if (step > 5000) return [];

    if (actives) {

      actives.sort((a, b) => diff(a.status, end) - diff(b.status, end););

      const { status, name } = actives[0];

      const index = status.findIndex((item) => item === 0);

      const arr = [];

      for (let j = 1; j <= 4; j++) {

        const newStatus = moveAI(status, index, j);

        if (newStatus) {

          if (statuses.every((status) => diff(status, newStatus))) {

            statuses.push(newStatus);

            traces.push({ [statuses.length - 1]: name });

            arr.push({ status: newStatus, step, name: statuses.length - 1 });

            if (!diff(newStatus, end)) {

              out = true;

              break;

            }

          }

        }

      }

      actives.splice(0, 1, ...arr);

    } else break;

  }

  const res = [];

  let tr = Object.keys(traces[traces.length - 1])[0];

  while (traces.find((t) => t[tr])) {

    res.unshift(tr);

    tr = traces.find((t) => t[tr])[tr];

  }

  res.unshift(tr);

  res.unshift(0);

  return res.map((r) => statuses[r]);

}

* Di chuyển các thành phần trong status

function moveAI(st, i, dir) {

  const newSt = [...st];

  if (dir === 1) {

    if (i % 3 === 0) return null;

    [newSt[i - 1], newSt[i]] = [newSt[i], newSt[i - 1]];

  } else if (dir === 2) {

    if ((i + 1) % 3 === 0) return null;

    [newSt[i + 1], newSt[i]] = [newSt[i], newSt[i + 1]];

  } else if (dir === 3) {

    if (!newSt[i - 3]) return null;

    [newSt[i - 3], newSt[i]] = [newSt[i], newSt[i - 3]];

  } else if (dir === 4) {

    if (!newSt[i + 3]) return null;

    [newSt[i + 3], newSt[i]] = [newSt[i], newSt[i + 3]];

  }

  return newSt;

}

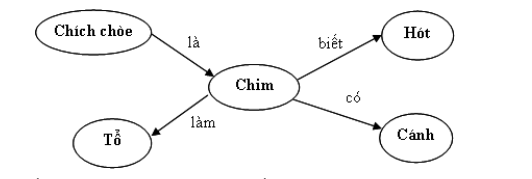
1. Giải bài toán tam giác bằng biểu diễn tri thức mạng ngữ nghĩa
2. Lý thuyết

Mạng ngữ nghĩa là một phương pháp biểu diễn tri thức đầu tiên và cũng là phương pháp dễ hiểu nhất đối với chúng ta. Phương pháp này sẽ biểu diễn tri thức dưới dạng một đồ thị, trong đó đỉnh là các đối tượng (khái niệm) còn các cung cho biết mối quan hệ giữa các đối tượng (khái niệm) này.

Chẳng hạn: giữa các khái niệm chích chòe, chim, hót, cánh, tổ có một số mối quan hệ như sau:

* Chích chòe là một loài chim.
* Chim biết hót
* Chim có cánh
* Chim sống trong tổ

Các mối quan hệ này sẽ được biểu diễn trực quan bằng một đồ thị như sau:



Một mạng ngữ nghĩa bao gồm một tập các nút và các liên kết để biểu diễn định nghĩa của một (tập) khái niệm

* Các nút biểu diễn các khái niệm
* Các liên kết biểu diễn các mối quan hệ giữa các khái niệm

Quá trình suy diễn trong mạng ngữ nghĩa được thực hiện thông qua cơ chế lan truyền

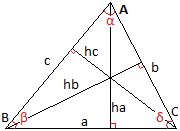
* Tác động (Activation)
* Kế thừa (Inheritance)

1. Giải bài toán tam giác bằng biểu diễn tri thức mạng ngữ nghĩa:

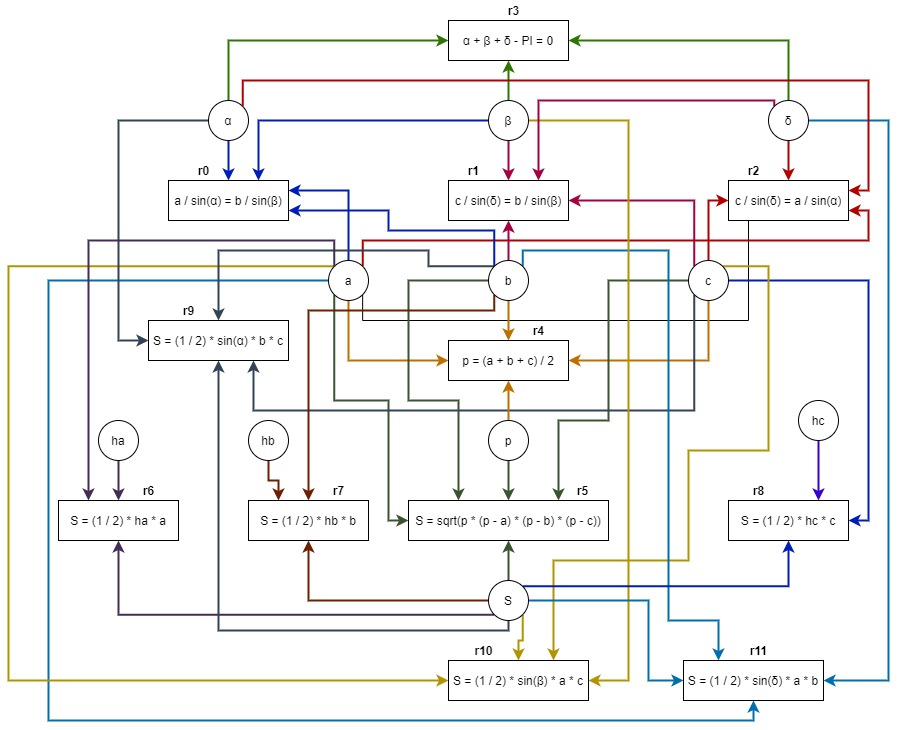
* Có 22 yếu tố của tam giác. Như vậy có C322 - 1 cách để xây dựng hay xác định một tam giác.
* Theo thống kê, có khoảng 200 công thức liên quan đến cạnh và góc 1 tam giác.
* Để giải bài toán này bằng công cụ mạng ngữ nghĩa, sử dụng khoảng 200 đỉnh để chứa công thức và khoảng 22 đỉnh để chứa các yếu tố của tam giác. Mạng ngữ nghĩa cho bài toán này có cấu trúc như sau:
* Đỉnh của đồ thị gồm hai loại:
* Đỉnh chứa công thức (ký hiệu bằng hình chữ nhật)
* Đỉnh chứa yếu tố của tam giác (ký hiệu bằng hình tròn)
* Cung: chỉ nối từ đỉnh hình tròn đến đỉnh hình chữ nhật cho biết yếu tố tam giác xuất hiện trong công thức nào.

**Lưu ý:** Trong một công thức liên hệ giữa n yếu tố của tam giác, giả định rằng nếu đã biết giá trị của n-1 yếu tố thì sẽ tính được giá trị của yếu tố còn lại. Chẳng hạn như trong công thức tổng 3 góc của tam giác bằng 180o thì khi biết được hai góc, ta sẽ tính được góc còn lại.

Ở bài báo cáo này chỉ sử dụng một số yếu tố của tam giác và công thức.



* Các yếu tố:
* a: cạnh a
* b: cạnh b
* c: cạnh c
* α: góc alpha
* β: góc beta
* δ: góc delta
* p: nữa chu vi tam giác
* ha: đường cao từ đỉnh A
* hb: đường cao từ đỉnh B
* hc: đường cao từ đỉnh C
* S: diện tích tam giác
* Các công thức:
* r0: a / sin(α) = b / sin(β)
* r1: c / sin(δ) = b / sin(β)
* r2: c / sin(δ) = a / sin(α)
* r3: α + β + δ - PI = 0
* r4: p = (a + b + c) / 2
* r5: S = sqrt(p \* (p - a) \* (p - b) \* (p - c))
* r6: S = (1 / 2) \* ha \* a
* r7: S = (1 / 2) \* hb \* b
* r8: S = (1 / 2) \* hc \* c
* r9: S = (1 / 2) \* sin(α) \* b \* c
* r10: S = (1 / 2) \* sin(β) \* a \* c
* r11: S = (1 / 2) \* sin(δ) \* a \* b
* Mạng ngữ nghĩa:

****

1. Thuật toán

Cơ chế suy diễn thực hiện theo thuật toán “loang” như sau:

* Bước 1: Kích hoạt những đỉnh hình tròn đã cho ban đầu (những yếu tố đã có giá trị)
* Bước 2: Lặp lại bước sau cho đến khi kích hoạt được tất cả những đỉnh ứng với những yếu tố cần tính hoặc không thể kích hoạt được bất kỳ đỉnh nào nữa.

Nếu một đỉnh hình chữ nhật có cung nối với n đỉnh hình tròn mà n-1 đỉnh hình tròn đã được kích hoạt thì kích hoạt đỉnh hình tròn còn lại (và tính giá trị đỉnh còn lại này thông qua công thức ở đỉnh hình chữ nhật).

1. Cài đặt

Cài đặt mạng ngữ nghĩa giải bài toán tam giác bằng một mảng hai chiều trong đó:

* Cột: ứng với công thức. Mỗi cột ứng với một công thức tam giác khác nhau (đỉnh hình chữ nhật).
* Dòng: ứng với yếu tố tam giác. Mỗi dòng ứng với một yếu tố tam giác khác nhau (đỉnh hình tròn).
* Phần tử A[i, j] = -1 nghĩa là trong công thức ứng với cột j có yếu tố tam giác ứng với cột i. Ngược lại A[i,j] = 0.

Để thực hiện thao tác “kích hoạt” một đỉnh hình tròn, ta đặt giá trị của toàn dòng ứng với yếu tố tam giác bằng 1.

Để kiểm tra xem một công thức đã có đủ n-1 yếu tố hay chưa (nghĩa là kiểm tra điều kiện “đỉnh hình chữ nhật có cung nối với n đỉnh hình tròn mà n-1 đỉnh hình tròn đã được kích hoạt”), lấy hiệu giữa tổng số ô có giá trị bằng 1 và tổng số ô có giá trị -1 trên cột ứng với công thức cần kiểm tra. Nếu kết quả bằng n, thì công thức đã có đủ n-1 yếu tố.

Sử dụng ngôn ngữ lập trình Javascript

* Mạng ngữ nghĩa gốc:

const ruleNetwork = [

// r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11

  [-1, 0, -1, 0, -1, -1, -1, 0, 0, 0, -1, -1], // A

  [-1, -1, 0, 0, -1, -1, 0, -1, 0, -1, 0, -1], // B

  [0, -1, -1, 0, -1, -1, 0, 0, -1, -1, -1, 0], // C

  [-1, 0, -1, -1, 0, 0, 0, 0, 0, -1, 0, 0], // a

  [-1, -1, 0, -1, 0, 0, 0, 0, 0, 0, -1, 0], // b

  [0, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, -1], //d

  [0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0], // p

  [0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0], // ha

  [0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0], // hb

  [0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0], // hc

  [0, 0, 0, 0, 0, -1, -1, -1, -1, -1, -1, -1], // S

]

* Kiểm tra 1 rule có thể được giải quyết hay không

function checkCanSolution(roleIndex, network) {

  // lấy rule từu trong network

  const rule = network.map(row => row[roleIndex]);

  // chỉ có 1 thành phần chưa biết trong rule này -> có thể giải quyết -> trả về index part tương tứng chưa biết

  if (rule.filter(item => item === -1).length === 1) {

    return rule.findIndex(item => item === -1);

  }

  // có nhiều hơn 1 thành phần chưa biết -> không thể giải quyết rule này

  return false;

}

* Cập nhật mạng ngữ nghĩa khi tim ra 1 thành phân nào đó

function fillNetWork(partIndex, network) {

  // thay đổi các vị trí chưa biết thành đã biết của partIndex trong network

  for (let i = 0; i < RULE\_COUNT; i++) {

    if (network[partIndex][i] === -1) {

      network[partIndex][i] = 1;

    }

  }

}

* Tìm lời giải cho bài toán

function solution(hypothesis, req) {

  // clone root network

  const network = ruleNetwork.map(row => [...row]);

  // chỉnh sửa network thông qua các giá trị giả thuyết

  const partHypothesis = Object.entries(hypothesis).filter(x => x[1]).map(y => y[0]);

  partHypothesis.forEach(part => fillNetWork(partIndex[part], network));

  // lấy ra các giá trị từ giả thuyết cùng các giá trị chưa biết

  let { A, B, C, a, b, d, p, ha, hb, hc, S } = hypothesis;

  // part yêu cầu tìm kiếm

  // index của part yêu cầu tìm kiếm trong network

  const row = partIndex[req]

  // lưu vết

  const traces = [];

  // tìm kiếm giá trị yêu cầu

  while(!network[row].every(item => item !== -1)) {

    // sự thay đổi

    let hasChange = false;

    // duyệt lần lượt các rule, rule nào có thể giải quyết thì giải quyết

    for (let ruleIndex = 0; ruleIndex < RULE\_COUNT; ruleIndex++) {

      // kiểm tra có thể giải quết rule thứ ruleIndex

      const part = checkCanSolution(ruleIndex, network);

      // rule ruleIndex có thể được giải quết

      if (part || part === 0) {

        let trace = ruleTexts[ruleIndex];

        // tính giá trị còn lại chưa biết của rule tương ứng

        const value = calcRestValue(ruleIndex, { A,B,C,a,b,d,p,ha,hb,hc,S });

        // gán giá trị vừa tính cho part tưng ứng

         switch(part) {

          case 0: A = value; trace += ` => a = ${round(value)}`; break;

          case 1: B = value; trace += ` => b = ${round(value)}`; break;

          case 2: C = value; trace += ` => c = ${round(value)}`; break;

          case 3: a = value; trace += ` => α = ${round(toDegrees(value))}°`; break;

          case 4: b = value; trace += ` => β = ${round(toDegrees(value))}°`; break;

          case 5: d = value; trace += ` => δ = ${round(toDegrees(value))}°`; break;

          case 6: p = value; trace += ` => p = ${round(value)}`; break;

          case 7: ha = value; trace += ` => ha = ${round(value)}`; break;

          case 8: hb = value; trace += ` => hb = ${round(value)}`; break;

          case 9: hc = value; trace += ` => hc = ${round(value)}`; break;

          case 10: S = value; trace += ` => S = ${round(value)}`; break;

          default: break;

        }

        traces.push(trace);

        // thay đôi network qua part vừa tính

        fillNetWork(part, network);

        hasChange = true; // có sự thay đổi

        // tìm được part yêu cầu

        if (part === row) break; // thoát -> OK

      }

    }

    // không có sự thay đổi network khi duyệt qua hết các rule

    if (!hasChange) throw new Error('Không đủ dữ kiện để tìm ra kết quả!');

  }

  return {

    traces, // vết

    result: { A, B, C, a, b, d, p, ha, hb, hc, S }[req] // giá trị cần tìm

  };

}

1. Giải thuật Quinlan
2. Lý thuyết

* Là thuật toán học theo quy nạp dùng luật, đa mục tiêu.
* Do Quinlan đưa ra năm 1979.
* Ý tưởng: Chọn thuộc tính quan trọng nhất để tạo cây quyết định.
* Thuộc tính quan trọng nhất là thuộc tính phân loại bảng quan sát thành các bảng con sao cho từ mỗi bảng con này dễ phân tích để tìm quy luật chung.

1. Thuật toán

* Bước 1: Phân loại thuộc tính dẫn xuất và thuộc tính mục tiêu (thuộc tính mục tiêu: là thuộc tính quan tâm, thuộc tính dẫn xuất: là thuộc tính quan quan sát)
* Bước 2: Tính độ bất định Entropy cho mỗi thuộc tính và chọn thuộc tính có Entropy nhỏ nhất theo công thức:
* Bước 3: Tính vector đặt trưng của thuộc tính vừa chọn

**VA(j) = (T(j , r1), T(j , r2) , …, T(j , rn))**

**T(j, ri)** = (tổng số phần tử trong phân hoạch có giá trị thuộc tính dẫn xuất **A** là **j** và có giá trị thuộc tính mục tiêu là **ri** ) / ( tổng số phần tử trong phân hoạch có giá trị thuộc tính dẫn xuất **A** là **j** )

\* **r1, r2, …, rn** là các giá trị của thuộc tính mục tiêu

* Bước 4: Loại bỏ thuộc tính đã được phân hoạch.
* Nếu vẫn còn thuộc tính đẫn xuất quay lại bước 2 để tính độ bất định Etropy cho các thuộc tính dẫn xuất.
* Ngược lại, kết thúc thuật toán.

1. Cài đặt thuật toán.

function quinlan(obj) {

  let rules = []; // tập luật

  let out = false;

  const keys = Object.keys(obj[0]); // danh sách trường khảo sát

  while (!out) {

    const entropies = []; // mảng kết quả tính entropy

    // tính entropy

    keys.forEach((key, index) => {

      if (key.toLowerCase() !== 'stt' && index < keys.length - 1) {

        entropies.push({key, entropy: entropy(obj.map(row => [row[key], row[keys[keys.length - 1]]]))});

      }

    });

    // sắp xếp mảng entropy để tìm phần tử có entropy nhỏ nhất

    entropies.sort((a, b) => a.entropy - b.entropy);

    // tìm các vector của trường có entropy nhỏ nhất

    const vectors = vector(obj.map(row => [row[entropies[0].key], row[keys[keys.length - 1]]]));

    // tìm luật

    if (rules.length === 0) {

   // chưa có luật nào > khởi tạo luật: bắt đầu là trường có entropy nhỏ nhất

      vectors.forEach(vec => {

        const rule = { [entropies[0].key] : Object.keys(vec)[0] }

        // kiểm tra các vector của trường này có đưa ra luật hay không?

        Object.values(vec).forEach(v => {

          v.forEach(item => {

            if (Object.values(item)[0] === 1) {

              // Có thể kết luận -> đưa vào rule

              rule[keys[keys.length - 1]] = Object.keys(item)[0];

              // Xóa các thành phần đã đưa ra kết luật trong obj

              obj = obj.filter(row => {

                return row[entropies[0].key] !== Object.keys(vec)[0];

              })

            }

          })

        })

        rules.push(rule);

      });

    } else {

      // đã có rules

      // các rule chưa có kết luận

      const rulesFilter = rules.filter(rule => !(keys[keys.length - 1] in rule));

      // các rule đã có kết luật

      rules = rules.filter(rule => keys[keys.length - 1] in rule);

      // tìm và thêm các thành phần vào các rule chưa có kết luận -> như trên

      rulesFilter.forEach(rule => {

        vectors.forEach(vec => {

          const r = { ...rule, [entropies[0].key] : Object.keys(vec)[0] };

          Object.values(vec).forEach(v => {

            v.forEach(item => {

              if (Object.values(item)[0] === 1) {

                r[keys[keys.length - 1]] = Object.keys(item)[0];

                obj = obj.filter(row => {

                  return row[entropies[0].key] !== Object.keys(vec)[0];

                })

              }

            })

          })

          rules.push(r);

        });

      });

    }

    // nếu tất cả các rule đã đưa ra được kết luận -> kết thúc

    if (rules.every(rule => keys[keys.length - 1] in rule)) {

      out = true;

    }

  }

  // trả về dánh sách rule cùng key kết luận

  return {

    key: keys[keys.length - 1],

    rules

  };

}

// Hàm tính vector

function vector(data) {

  const set = new Set(data.map(row => row[1]));

  const { groups, subGroups } = createGroup(data);

  const vectors = [];

  subGroups.forEach(sub => {

    const vec = [];

    set.forEach(item => {

      vec.push({ [item]: (Object.values(sub)[0][item] || 0) / groups[Object.keys(sub)[0]].length })

    });

    vectors.push({ [Object.keys(sub)[0]]: vec });

  });

  return vectors;

}

// Hàm tính entropy

function entropy(data) {

  const groups = data.reduce((acc, cur) => {

    if (cur[0] in acc) {

      acc[cur[0]].push(cur[1])

    } else {

      acc[cur[0]] = [cur[1]];

    }

    return acc;

  }, {});

  let entropySum = 0;

  for (let group in groups) {

    const subGroups = groups[group].reduce((a, c) => {

      if (c in a) {

        a[c]++;

      } else {

        a[c] = 1;

      }

      return a;

    }, {});

    let subEntropy = 0;

    for (let subGroup in subGroups) {

      subEntropy += -(subGroups[subGroup] / groups[group].length) \* Math.log2(subGroups[subGroup] / groups[group].length);

    }

    entropySum += (groups[group].length / data.length) \* subEntropy;

  }

  return entropySum;

}

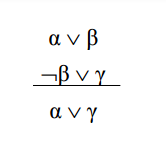
1. Giải thuật Robinson
2. Lí thuyết

Thuật toán Hợp giải (Robinson) – Hợp giải Robison – là phương pháp chứng minh phản chứng:

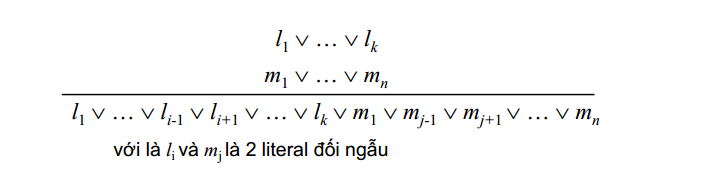
Muốn chứng minh  là đúng, ta Chứng minh điều ngược lại  là sai.

Hợp giải là đúng và đủ cho logic mệnh đề

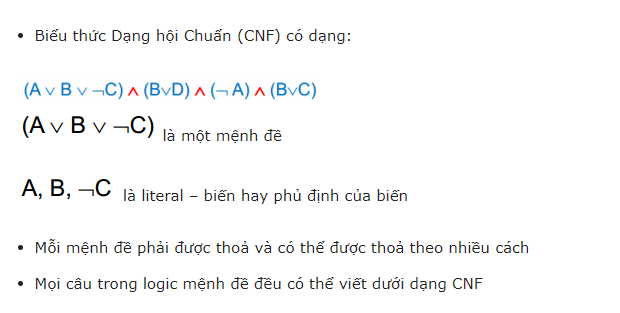
* **Hợp giải mệnh đề**

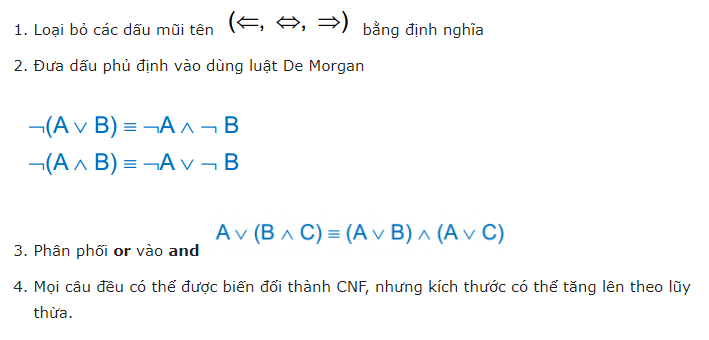
Hợp giải mệnh đề là phương pháp suy diễn dưa trên luật hợp giải

Phát biểu một cách tổng quát, ta có:



* Chỉ sử dụng một mình hợp giải mệnh đề (không cần sử dụng các luật khác) có thể xây dựng một chương trình chứng minh lý thuyết đúng và đủ cho tất cả logic mệnh đề
* Mệnh đề kết quả chỉ chứa một bản sao cho mỗi literal. Việc loại bỏ literal trùng gọi là Factoring.
* Đòi hỏi tất cả các câu được chuyển sang dạng hội chuẩn CNF (Conjunctive Normal Form)
* **Dạng hội chuẩn CNF**





1. Thuật toán

* Bước 1: Biến đổi tất cả các câu thành dạng CNF
* Bước 2: Lấy phủ kết luận, đưa vào KB
* Bước 3: Lặp
* Nếu trong KB có chứa hai mệnh đề phủ định nhau thì trả về **false // trả về false =>** **vô lý**
* Nếu có hai mệnh đề chứa các literal phủ định nhau thì áp dụng hợp giải.
* Lặp cho đến khi không thể áp dụng tiếp luật hợp giải.
* Bước 4: Trả về **true**

1. Cài đặt thuật toán

* Kiểm tra mệnh đề đối ngẫu

bool KiemTraDoiNgau(string str1, string str2){

int size\_str1 = str1.size();

int size\_str2 = str2.size();

if(size\_str1 > size\_str2){

str1.erase(0, 1);

if(str1.compare(str2) == 0)

return true;

} else

if(size\_str1 < size\_str2){

str2.erase(0, 1);

if(str2.compare(str1) == 0)

return true;

}

return false;

}

* Kiểm tra trong danh sách có mệnh đề đối ngẫu hay không

bool KiemTraDoiNgauDS(){

int i, j = 0;

int size\_i = ds2.size();

int size\_j = size\_i - 1;

list<string> ds3 = ds2;

list<string> :: iterator it;

list<string> :: iterator ot;

for(it = ds2.begin(); it != ds2.end(); ++it){

string s1 = \*it;

ds3.pop\_front();

for(ot = ds3.begin(); ot != ds3.end(); ++ot){

string s2 = \*ot;

if(KiemTraDoiNgau(s1, s2))

return true;

}

}

return false;

}

* Phép hợp giải

string HopGiai3(string str1, string str2){

string s = str1;

string s1, s2;

s1 = s.substr(1, s.find("v") - 1);

s.erase(0, s.find("v") + 1);

s2 = s.substr(0, s.find(")"));

if(KiemTraDoiNgau(s1, str2))

return s2;

if(KiemTraDoiNgau(s2, str2))

return s1;

// khong the hop giai

return str1;

}

* Phép hợp giải

string HopGiai4(string str1, string str2){

string s1 = str1;

string s1\_1, s1\_2;

s1\_1 = s1.substr(1, s1.find("v") - 1);

s1.erase(0, s1.find("v") + 1);

s1\_2 = s1.substr(0, s1.find(")"));

string s2 = str2;

string s2\_1, s2\_2;

s2\_1 = s2.substr(1, s2.find("v") - 1);

s2.erase(0, s2.find("v") + 1);

s2\_2 = s2.substr(0, s2.find(")"));

if (KiemTraDoiNgau(s1\_1, s2\_1)){

string s;

s.insert(s.size(), "(");

s.insert(s.size(), s1\_2);

s.insert(s.size(), "v");

s.insert(s.size(), s2\_2);

s.insert(s.size(), ")");

return s;

}

if(KiemTraDoiNgau(s1\_1, s2\_2)){

string s;

s.insert(s.size(), "(");

s.insert(s.size(), s1\_2);

s.insert(s.size(), "v");

s.insert(s.size(), s2\_1);

s.insert(s.size(), ")");

return s;

}

if(KiemTraDoiNgau(s1\_2, s2\_1)){

string s;

s.insert(s.size(), "(");

s.insert(s.size(), s1\_1);

s.insert(s.size(), "v");

s.insert(s.size(), s2\_2);

s.insert(s.size(), ")");

return s;

}

if(KiemTraDoiNgau(s1\_2, s2\_2)){

string s;

s.insert(s.size(), "(");

s.insert(s.size(), s1\_1);

s.insert(s.size(), "v");

s.insert(s.size(), s2\_1);

s.insert(s.size(), ")");

return s;

}

string s;

s.insert(s.size(), "88888888");

// khong the hop giai

return s;

}

* Chuyển đổi giả thuyết vào thành dạng chuẩn Robinson

void ChuanHoa(){

string s;

// doc file dua vao 2 list

fi.open("data.txt", ios::in);

getline(fi, s, '>');

cout << s << endl;

s.insert(s.size(), "^");

while(!s.empty()){

ds1.push\_back(s.substr(0, s.find("^")));

s.erase(0, s.find("^") + 1);

}

getline(fi, s);

cout << s << endl;

s.insert(s.size(), "v");

while(!s.empty()){

ds2.push\_back(s.substr(0, s.find("v")));

s.erase(0, s.find("v") + 1);

}

fi.close();

// xu ly chuan hoa ds2

for(int i = 0; i < ds2.size(); i++){

string s2;

if(ds2.front().size() == 2){

s2 = ds2.front().erase(0, 1);

} else

if(ds2.front().size() == 1){

s2 = ds2.front().insert(0, "-");

}

ds2.pop\_front();

ds2.push\_back(s2);

}

}

* Xử lý Robinson

void XuLyRoBinSon(){

// xu ly theo phep hop giai: a ^ (-a v b) => b

while (!KiemTraDoiNgauDS()){

string s1 = ds1.front();

string s2 = ds2.front();

string s = HopGiai3(s1,s2);

cout << s1 << " ^ " << s2 << " -> " << s << endl;

if (s.size() <= 2){

ds1.pop\_front();

ds2.pop\_front();

ds2.push\_back(s);

} else {

ds1.push\_back(ds1.front());

ds1.pop\_front();

}

}

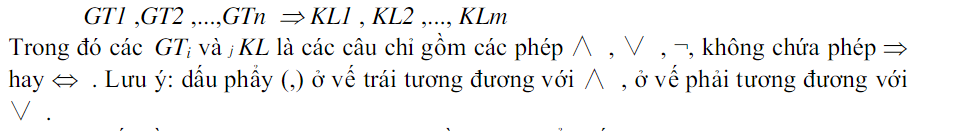
}

1. Giải thuật vương hạo
2. Lí thuyết

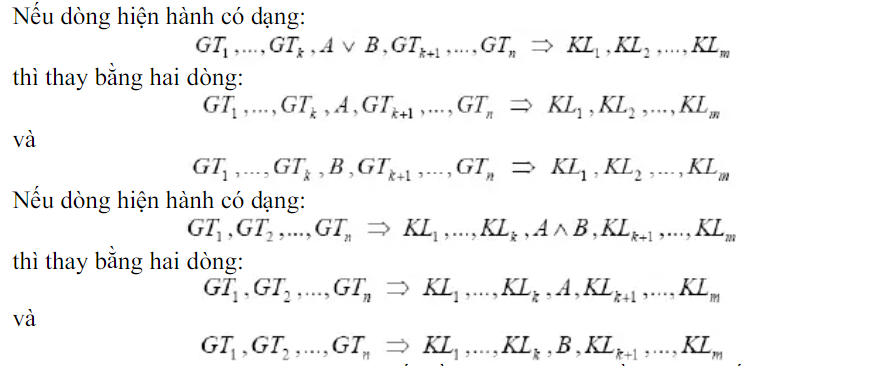
Thuật toán Vương Hạo là một thuật giải khác để chứng minh việc suy dẫn mệnh đề ( có cùng mục đích như Robinson).

1. Thuật toán

* Bước 1: Đưa bài toán cần chứng minh về dạng chuẩn:



* Bước 2: Nếu tồn tại một cậu có phép ¬ ở đầu thì chuyển vế câu và loại bỏ phép ¬
* Bước 3: Thay các dấu ˄ ở vế trái và các dấu ˅ ở vế phải bằng dấu phẩy (,). Khi đó, vế trái chỉ còn dấu ˅ và ¬, vế phải chỉ còn dấu ˄ và ¬.
* Bước 4:



* Bước 5: Một dòng chứng minh nếu tồn tại một mệnh đề ở cả hai vế.
* Bước 6: Một dòng không thể tách, cũng không thể chuyển dấu ¬ mà không có biến mệnh chung ở cả hai vế thì không được chứng minh.

Lặp lại bước 2 đến 6 cho tới khi mọi dòng được chứng minh hay tồn tại một dòng không được chứng minh. Bài toán ban đầu được chứng minh nếu mọi dòng tách ra từ nó được chứng minh.

1. Cài đặt thuật toán

from abc import abstractmethod

import copy

class val:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return self.name

def \_\_eq\_\_(self, other):

try:

if self.name == other.name:

return True

except:

return False

return False

def isval(self):

return True

def istuyen(self):

return False

def ishoi(self):

return False

def isphu(self):

return False

class menhde:

def \_\_init\_\_(self, val1, val2):

pass

def \_\_str\_\_(self):

pass

def istuyen(self):

return False

def ishoi(self):

return False

def isphu(self):

return False

@abstractmethod

def \_\_eq\_\_(self, other):

# Phải cùng kiểu hội hoặc tuyển

condition1 = self.istuyen() and other.istuyen()

if not condition1:

return False

# 2 vế bằng nhau

condition2 = self.val1 == other.val1 and self.val2 == other.val2

condition3 = self.val1 == other.val2 and self.val2 == other.val1

if condition2 or condition3:

return True

return False

class hoi(menhde):

def \_\_init\_\_(self, val1, val2):

self.val1 = val1

self.val2 = val2

def \_\_str\_\_(self):

return "(" + str(self.val1) + "^" + str(self.val2) + ")"

def ishoi(self):

return True

def phanra(self): # a^b chuyển thành a,b

return [self.val1, self.val2]

class tuyen(menhde):

def \_\_init\_\_(self, val1, val2):

self.val1 = val1

self.val2 = val2

def \_\_str\_\_(self):

return "(" + str(self.val1) + "v" + str(self.val2) + ")"

def istuyen(self):

return True

def phanra(self): # avb chuyển thành a,b

return [self.val1, self.val2]

class phu(menhde):

def \_\_init\_\_(self, val):

self.val = val

def \_\_str\_\_(self):

return "!" + str(self.val)

def isphu(self):

return True

def \_\_eq\_\_(self, other):

try:

if self.val == other.val:

return True

except:

return False

return False

def isval(self):

return True

class bieuthuc:

# Bước 1: Khởi tạo với dạng chuẩn

def \_\_init\_\_(self, vetrai, vephai, prev=None):

self.prev = prev # Node biểu thức trước đó

# Nhận vào các mệnh đề vế trái và vế phải theo dạng list

self.vetrai = vetrai # Vế trái là 1 list chứa các mệnh đề

self.vephai = vephai # Vế phải là 1 list chứa các mệnh đề

# Bước 2: Chuyển vế đổi dấu

def chuyenve(self):

flag = 0

temp = copy.deepcopy(self)

for menhde in self.vetrai:

if menhde.isphu():

self.vephai.append(menhde.val) # Đưa qua vế phải, bỏ phủ

self.vetrai.remove(menhde) # Xóa khỏi vế trái

flag = 1

for menhde in self.vephai:

if menhde.isphu():

self.vetrai.append(menhde.val) # Đưa qua vế trái, bỏ phủ

self.vephai.remove(menhde) # Xóa khỏi vế phải

flag = 1

# Bước 3: Phân rã hội của giả thiết và tuyển của kết luận thành dấu phẩy

def phanra(self):

flag = 0

for menhde in self.vetrai: # Các phép tính ở vế trái

if menhde.ishoi(): # Nếu là hội

self.vetrai += menhde.phanra() # Phân rã thành 2 mệnh đề con

self.vetrai.remove(menhde) # Xóa mệnh đề cũ

flag = 1

for menhde in self.vephai: # Các phép tính ở vế phải

if menhde.istuyen(): # Nếu là tuyển

self.vephai += menhde.phanra() # Phân rã thành 2 mệnh đề con

self.vephai.remove(menhde) # Xóa mệnh đề cũ

flag = 1

if flag == 1:

print('Phân rã: ', self)

else:

print('Không phân rã được:', self)

if self.checkcm() == 1:

print(' (ĐCM)')

elif self.checkcm() == -1:

print(' (Ko ĐCM)')

else:

print('\n')

# Hàm kiểm tra trạng thái "Được chứng minh", "Ko được chứng minh", "Còn chứng minh"

def checkcm(self):

self.chuyenve()

# 1 là được chứng minh, 0 là còn chứng minh, -1 là không được chứng minh

for menhdevt in self.vetrai:

for menhdevp in self.vephai:

if menhdevt == menhdevp: # Nếu có 2 phần tử giống nhau -> dc chứng minh

return 1

donvi = True # Check xem tất cả đã trở thành dạng

try:

for menhdevt in self.vetrai:

if not menhdevt.isval(): # (Nếu không phải đơn vị) -> Xuất hiện exception

pass

except:

donvi = False

if donvi:

return -1

return 0

# Hàm chuyển thành 2 trạng thái

def chuyentrangthai(self):

tt1 = None

tt2 = None

temp = copy.deepcopy(self)

temp.prev = self

# Bước 1: phân rã

temp.phanra()

# Bước 2: Chuyển về

temp.chuyenve()

# Bước 3: Chuyển trạng thái

for menhde in self.vetrai: # Nếu vế trái

if menhde.istuyen(): # Có mệnh đề tuyển

temp.vetrai.remove(menhde)

tt1 = copy.deepcopy(temp)

tt1.vetrai.append(menhde.phanra()[0]) # Chuyển trạng thái 1

tt2 = copy.deepcopy(temp)

tt2.vetrai.append(menhde.phanra()[1]) # Chuyển trạng thái 2

return [tt1, tt2]

# Bước 3: Chuyển trạng thái

for menhde in self.vephai: # Nếu vế phải

if menhde.ishoi(): # Có mệnh đề hội

temp.vephai.remove(menhde)

tt1 = copy.deepcopy(temp)

tt1.phai.append(menhde.phanra()[0]) # Chuyển trạng thái 1

tt2 = copy.deepcopy(self)

tt2.phai.append(menhde.phanra()[1]) # Chuyển trạng thái 2

return [tt1, tt2]

return [tt1, tt2] # [None,None]

def \_\_str\_\_(self):

vetraistr = ','.join([str(menhde) for menhde in self.vetrai])

vephaistr = ','.join([str(menhde) for menhde in self.vephai])

return vetraistr + " => " + vephaistr

class VuongHao:

def \_\_init\_\_(self, bieuthuc):

self.Nodes = [bieuthuc]

def checktt(self): # Kiểm tra thử bài toán của mình đã được giải chưa

if -1 in [bieuthuc.checkcm() for bieuthuc in self.Nodes]:

return -1

if 0 in [bieuthuc.checkcm() for bieuthuc in self.Nodes]:

return 0

return 1

def run(self):

print('Biểu thức của chúng ta là: ', self.Nodes[0])

while (self.checktt() == 0):

for bieuthuc in self.Nodes:

self.Nodes.remove(bieuthuc)

tt1, tt2 = bieuthuc.chuyentrangthai()

if (tt1 == None):

continue

else:

self.Nodes.append(tt1)

self.Nodes.append(tt2)

print('Tách', str(bieuthuc), 'thành', tt1, ' và ', tt2)

if self.checktt() == -1:

break

print('KẾT QUẢ:', end='')

if self.checktt() == -1: print(' Không được chứng minh')

if self.checktt() == 1: print(' Được chứng minh')

for bieuthuc in self.Nodes:

print(bieuthuc)

# {a->b,b->c} => a->c

menhde1 = tuyen(phu(val('a')), val('b'))

menhde2 = tuyen(phu(val('b')), val('c'))

menhde3 = tuyen((val('a')), val('c'))

mybieuthuc = bieuthuc([menhde1, menhde2], [menhde3])

VuongHao(mybieuthuc).run()

1. Thuật toán di truyền.
2. Lí thuyết

Thuật toán di truyền là thuật toán tối ưu ngẫu nhiên dựa trên cơ chế chọn lọc tự nhiên và tiến hóa di truyền. Nguyên lý cơ bản của thuật toán di truyền đã được Holland giới thiệu vào năm 1962. Cơ sở toán học đã được phát triển từ cuối những năm 1960 và đã được giới thiệu trong quyển sách đầu tiên của Holland, Adaptive in Natural and Artificial Systems.

Thuật toán di truyền gồm có bốn quy luật cơ bản là lai ghép, đột biến, sinh sản và chọn lọc tự nhiên như sau:

* **Quá trình lai ghép (phép lai):**

Quá trình này diễn ra bằng cách ghép một hay nhiều đoạn gen từ hai nhiễm sắc thể cha-mẹ để hình thành nhiễm sắc thể mới mang đặc tính của cả cha lẫn mẹ. Phép lai này có thể mô tả như sau: Chọn ngẫu nhiên hai hay nhiều cá thể trong quần thể. Giả sử chuỗi nhiễm sắc thể của cha và mẹ đều có chiều dài là m. Tìm điểm lai bằng cách tạo ngẫu nhiên một con số từ 1 đến m-1. Như vậy, điểm lai này sẽ chia hai chuỗi nhiễm sắc thể cha-mẹ thành hai nhóm nhiễm sắc thể con là m1 và m2. Hai chuỗi nhiễm sắc thể con lúc này sẽ là m11+m22 và m21+m12. Đưa hai chuỗi nhiễm sắc thể con vào quần thể để tiếp tục tham gia quá trình tiến hóa.

* **Quá trình đột biến (phép đột biến):**

Quá trình tiến hóa được gọi là quá trình đột biến khi một hoặc một số tính trạng của con không được thừa hưởng từ hai chuỗi nhiễm sắc thể cha-mẹ. Phép đột biến xảy ra với xác suất thấp hơn rất nhiều lần so với xác suất xảy ra phép lai. Phép đột biến có thể mô tả như sau: Chọn ngẫu nhiên một số k từ khoảng 1 ≥ k ≥ m. Thay đổi giá trị của gen thứ k Đưa nhiễm sắc thể con vào quần thể để tham gia quá trình tiến hóa tiếp theo.

* **Quá trình sinh sản và chọn lọc (phép tái sinh và phép chọn):**

***Phép tái sinh:*** là quá trình các cá thể được sao chép dựa trên độ thích nghi của nó. Độ thích nghi là một hàm được gán các giá trị thực cho các cá thể trong quần thể của nó. Phép tái sinh có thể mô phỏng như sau: Tính độ thích nghi của từng cá thể trong quần thể, lập bảng cộng dồn các giá trị thích nghi đó (theo thứ tự gán cho từng cá thể) ta được tổng độ thích nghi. Giả sử quần thể có n cá thể. Gọi độ thích nghi của cá thể thứ i là Fi, tổng dồn thứ i là Ft. Tổng độ thích nghi là Fm. Tạo số ngẫu nhiên F có giá trị trong đoạn từ 0 đến Fm Chọn cá thể k đầu tiên thỏa mãn F ≥ Ft đưa vào quần thể của thế hệ mới.

***Phép chọn:*** là quá trình loại bỏ các cá thể xấu và để lại những cá thể tốt. Phép chọn được mô tả như sau: Sắp xếp quần thể theo thứ tự độ thích nghi giảm dần Loại bỏ các cá thể cuối dãy, chỉ để lại n cá thể tốt nhất.

1. Thuật toán

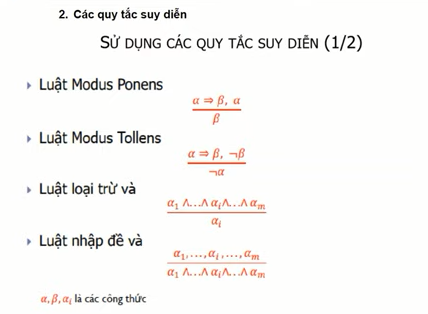
* Bước 1: Khởi tạo quần thể các nhiễm sắc thể.
* Bước 2: Xác định giá trị thích nghi của từng nhiễm sắc thể.
* Bước 3: Sao chép lại các nhiễm sắc thể dựa vào giá trị thích nghi của chúng và tạo ra những nhiễm sắc thể mới bằng phép toán di truyển.
* Bước 4: Loại bỏ những thành viên không thích nghi trong quần thể.
* Bước 5: Chèn những nhiễm sắc thể mới vào quẩn thể để hình thành một quần thể mới.
* Bước 6: Nếu mục tiêu tiềm kiếm đạt được thì dừng lại, ngược lại thì quay lại bước 3.

1. Cài đặt thuật toán.
2. Thuật toán suy diễn tiến
3. Lí thuyết

**Suy diễn tiến (lập luận tiến - forward chaining hoặc forward reasoning)** - tư tưởng cơ bản của suy diễn tiến là áp dụng luật suy diễn Modus Ponens tổng quát:

Là quá trình suy diễn bắt đầu từ tập sự kiện đã biết, rút ra những sự kiện mới và cứ như vậy cho đến khi có được sự kiện cần chứng minh hoặc không có luật nào sinh ra các sự kiện mới (tập sự kiện đúng là cực đại).

* **Các qui tắc suy diễn:**

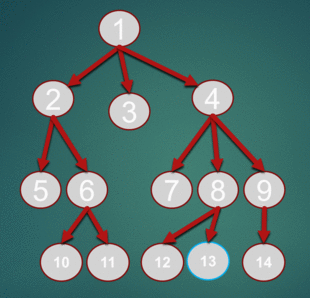


1. Thuật toán

Cơ sở: Bắt đầu từ những câu trong KB, sử dụng GMP và các qui tắc suy diễn để sinh ra các câu mới cho đến khi sinh không thể tạo ra các câu nào nữa.

1. Cài đặt thuât toán
2. Cài đặt thuật toán duyệt chiều rộng (BFS)
3. Lí thuyết

Trong lý thuyết đồ thị, tìm kiếm theo chiều rộng (Breath-First Search - BFS) là một thuật toán tìm kiếm trong đồ thị trong đó việc tìm kiếm chỉ bao gồm 2 thao tác: (a) cho trước một đỉnh của đồ thị; (b) thêm các đỉnh kề với đỉnh vừa cho vào danh sách có thể hướng tới tiếp theo. Có thể sử dụng thuật toán tìm kiếm theo chiều rộng cho hai mục đích: tìm kiếm đường đi từ một đỉnh gốc cho trước tới một đỉnh đích, và tìm kiếm đường đi từ đỉnh gốc tới tất cả các đỉnh khác. Trong đồ thị không có trọng số, thuật toán tìm kiếm theo chiều rộng luôn tìm ra đường đi ngắn nhất có thể. Thuật toán BFS bắt đầu từ đỉnh gốc và lần lượt nhìn các đỉnh kề với đỉnh gốc. Sau đó, với mỗi đỉnh trong số đó, thuật toán lại lần lượt nhìn trước các đỉnh kề với nó mà chưa được quan sát trước đó và lặp lại.



**Tính hiệu quả:** Hiệu quả khi lời giải nằm gần gốc của cây tìm kiếm. Hiệu quả của chiến lược phụ thuộc vào độ sâu của lời giải. Lời giải càng xa gốc thì hiệu quả của chiến lược càng giảm. Thuận lợi khi muốn tìm nhiều lời giải.

**Lượng bộ nhớ sử dụng để lưu trữ các trạng thái:** Phải lưu toàn bộ các trạng thái.

**Trường hợp xấu nhất:** Vét cạn toàn bộ.

**Trường hợp tốt nhất:** Lời giải được xác định khi mở nút đầu tiên.

1. Thuật toán

Thực hiện theo nguyên tắc hàng đợi – vào trước ra trước

* Bước 1: Đặt danh sách mở OPEN chứa trạng thái khởi đầu
* Bước 2: Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong danh sách mở OPEN, thực hiện:
* Chọn trạng thái (Ti) đầu tiên trong OPEN (và xóa Ti khỏi OPEN)
* Nếu T là trạng thái kết thúc thì thoát.
* Ngược lại, tạo ra các trạng thái kế tiếp Tk có thể có từ trạng thái T. Đối với mỗi trạng thái kế tiếp Tk thực hiện: g(Tk) = g(Ti) + cost (Ti, Tk); thêm Tk vào OPEN.

1. Cài đăt thuật toán

function duyetChieuRong(matrix, start, end) {

  start--; // do đồ thị bắt đầu từ 0

  end--; // do đồ thị bắt đầu từ 0

  // active: [ [a, b] ] -> a: điểm, b: chi phí

  const actives = [ [start, 0] ];

  // traces: [ { a: b } ] -> a: điểm được mở, b: điểm mở

  const traces = [];

  let out = false;

  while(!out && actives.length) {

    // hàng đợi -> chiều rộng

    // lấy và xóa điểm đàu tiên ra khỏi actives

    const node = actives.shift();

    // từ điểm lấy ra mở tới những điểm có thể

    for (let i = 0; i < matrix[0].length; i++) {

      if (matrix[node[0]][i] !== 0) {

        // thêm điểm mới vòa actives

        actives.push([i, node[1] + matrix[node[0]][i]]);

        // thêm vết mới vào traces

        traces.push({ [i]: node[0] });

        if (i === end) {

          // tới đích

          out = true;

          break;

        }

      }

    }

  }

  if (out) {

    // truy vết

    const dir = [];

    let tr = Object.keys(traces[traces.length - 1])[0];

    while (traces.find((t) => t[tr])) {

      dir.unshift(+tr + 1);

      tr = traces.find((t) => t[tr])[tr];

    }

    dir.unshift(+tr + 1);

    dir.unshift(start + 1);

    return {

      f: actives.pop()[1], // tổng chi phí

      dir // vết

    }

  }

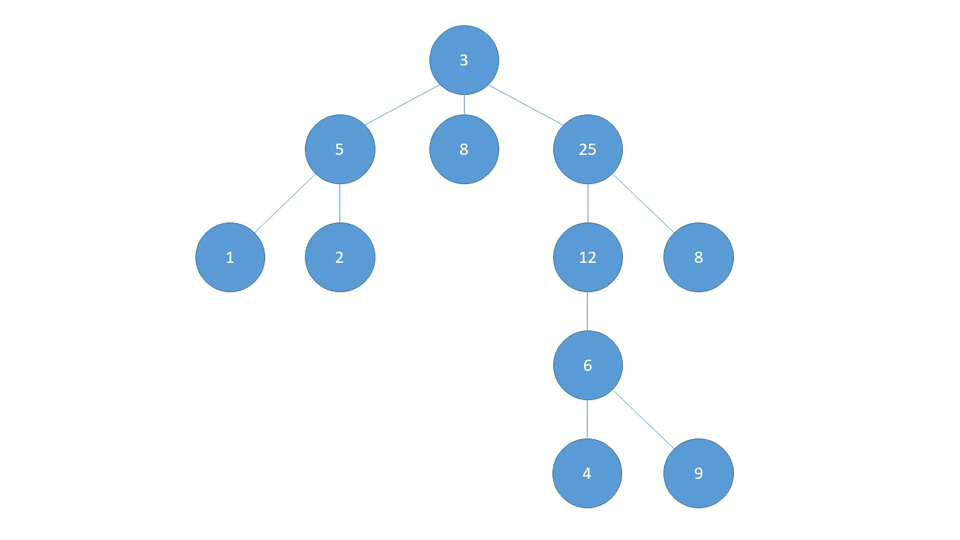
  return null; // không tìm được đường đi

}

1. Cài đặt thuật toán duyệt chiều sâu (DFS)
2. Lí thuyết

Tìm kiếm theo chiều sâu (Depth-first search - DFS) là một thuật toán duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị. Thuật toán khởi đầu tại gốc (hoặc chọn một đỉnh nào đó coi như gốc) và phát triển xa nhất có thể theo mỗi nhánh.

Thông thường, DFS là một dạng tìm kiếm thông tin không đầy đủ mà quá trình tìm kiếm được phát triển tới đỉnh con đầu tiên của nút đang tìm kiếm cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước. Trong dạng không đệ quy, tất cả các đỉnh chờ được phát triển được bổ sung vào một ngăn xếp LIFO.



1. Thuật toán

Thực hiện theo nguyên tắc ngăn xếp – vào sau ra trước (vào sau ra trước)

* Bước 1: Đặt OPEN chứa trạng thái khởi đầu.
* Bước 2: Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong danh sách mở OPEN, thực hiện:
* Chọn trạng thái (Ti) cuối cùng trong OPEN (và xóa Ti khỏi OPEN)
* Nếu T là trạng thái kết thúc thì thoát.
* Ngược lại, tạo ra các trạng thái kế tiếp Tk có thể có từ trạng thái T. Đối với mỗi trạng thái kế tiếp Tk thực hiện: g(Tk) = g(Ti) + cost (Ti, Tk); thêm Tk vào OPEN.

1. Cài đặt thuật toán

function duyetChieuSau(matrix, start, end) {

  start--; // do đồ thị bắt đầu từ 0

  end--; // do đồ thị bắt đầu từ 0

  // active: [ [a, b] ] -> a: điểm, b: chi phí : Open

  const actives = [ [start, 0] ];

  // traces: [ { a: b } ] -> a: điểm được mở, b: điểm mở

  const traces = [];

  let out = false;

  while(!out && actives.length) {

    // ngăn xếp -> chiều sâu

    // lấy và xóa điểm cuối cùng ra khỏi actives

    const node = actives.pop();

    // từ điểm lấy ra mở tới những điểm có thể

    for (let i = 0; i < matrix[0].length; i++) {

      if (matrix[node[0]][i] !== 0) {

        // thêm điểm mới vào actives

        actives.push([i, node[1] + matrix[node[0]][i]]);

        // thêm vết mới vào traces

        traces.push({ [i]: node[0] });

        if (i === end) {

          // tới đích

          out = true; break;

        }

      }

    }

  }

  if (out) {

    // truy vết

    const dir = [];

    let tr = Object.keys(traces[traces.length - 1])[0];

    while (traces.find((t) => t[tr])) {

      dir.unshift(+tr + 1);

      tr = traces.find((t) => t[tr])[tr];

    }

    dir.unshift(+tr + 1);

    dir.unshift(start + 1);

    return {

      f: actives.pop()[1], // tổng chi phí

      dir // vết

    }

  }

  return null; // không tìm được đường đi

}

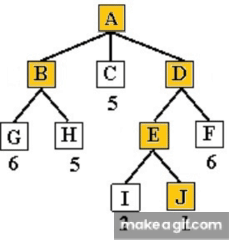
1. Tìm kiếm Heuristic bằng tìm kiếm ưu tiên tối ưu
2. Lí thuyết

Tìm kiếm ưu tiên tối ưu (Best-first search - BFS) là một thuật toán tìm kiếm tối ưu hóa tìm kiếm theo chiều rộng bằng cách mở rộng nút hứa hẹn nhất được chọn theo một quy tắc nào đó.

Tìm kiếm theo lựa chọn tốt nhất là việc ước lượng mức độ hứa hẹn của nút n theo một “hàm đánh giá heuristic f(n). Hàm này nói chung có thể phụ thuộc vào mô tả của n, mô tả về điểm đích, thông tin thu thập được bởi quá trình tìm kiếm cho tới thời điểm đó, và quan trọng nhất là phụ thuộc vào mọi tri thức bổ sung về miền xác định của bài toán.”

Để có được hiệu quả về thời gian chạy cho việc chọn ra ứng cử viên tốt nhất cho việc mở rộng, người ta thường dùng một hàng đợi ưu tiên để cài đặt cấu trúc dữ liệu lưu trữ các lựa chọn hiện hành.

Ví dụ về các thuật toán tìm kiếm ưu tiên tối ưu là thuật toán Dijkstra và giải thuật tìm kiếm A\*. Các thuật toán tìm kiếm ưu tiên tối ưu thường được sử dụng để tìm đường trong quá trình tìm kiếm tổ hợp.



1. Thuật toán

Thuật giải AT là một phương pháp tìm kiếm theo kiểu BFS với độ tốt của nút là giá trị hàm g tổng chiều dài con đường đã đi từ trạng thái bắt đầu đến trạng thái hiện tại

Thuật giải AT

* Bước 1: Đặt OPEN chứa trạng thái khởi đầu.
* Bước 2: Cho đến khi tìm được trạng thái đích hoặc không còn nút nào trong OPEN, thực hiện:
* Chọn trạng thái (Tmax) có giá trị g nhỏ nhất trong OPEN (và xóa Tmax khỏi OPEN)
* Nếu Tmax là trạng thái kết thúc thì thoát.
* Ngược lại, tạo ra các trạng thái kế tiếp Tk có thể có từ trạng thái Tmax. Đối với mỗi trạng thái kế tiếp Tk thực hiện: g(Tk) = g(Tmax) + cost(Tmax, Tk); thêm Tk vào OPEN.

1. Cài đặt thuật toán

function heuristic(matrix, start, end) {

  start--; // do đồ thị bắt đầu từ 0

  end--; // do đồ thị bắt đầu từ 0

  // active: [ [a, b] ] -> a: điểm, b: chi phí : Open

  const actives = [ [start, 0] ];

  // traces: [ { a: b } ] -> a: điểm được mở, b: điểm mở

  const traces = [];

  let out = false;

  while(!out && actives.length) {

    // tìm điểm có chi phí thấp nhất -> heuristic

    // 1. Tìm chi phí thấp nhất

    const min = Math.min(...actives.map(item => item[1]));

    // 2. lấy có chi phí = min và xóa điểm đó ra khỏi actives

    const indexMin = actives.findIndex(item => item[1] === min);

    const node = actives.splice(indexMin, 1)[0];

    // từ điểm lấy ra mở tới những điểm có thể

    for (let i = 0; i < matrix[0].length; i++) {

      if (matrix[node[0]][i] !== 0) {

        // thêm điểm mới vòa actives

        actives.push([i, node[1] + matrix[node[0]][i]]);

        // thêm vết mới vào traces

        traces.push({ [i]: node[0] });

        if (i === end) {

          // tới đích

          out = true; break;

        }

      }

    }

  }

  if (out) {

    // truy vết

    const dir = [];

    let tr = Object.keys(traces[traces.length - 1])[0];

    while (traces.find((t) => t[tr])) {

      dir.unshift(+tr + 1);

      tr = traces.find((t) => t[tr])[tr];

    }

    dir.unshift(+tr + 1);

    dir.unshift(start + 1);

    return {

      f: actives.pop()[1], // tổng chi phí

      dir // vết

    }

  }

  return null; // không tìm được đường đi

}