Louis Gacôgne

Intelligence artificielle

Cours, exercices et projets

2e édition augmentée

ellipses

INTRODUCTION

Définitions

Le terme d'intelligence artificielle (IA) a été introduit par Mc Carthy du MIT vers 1965.

« L'IA est la partie de l'informatique consacrée à la conception de systèmes informatiques intelligents » (E. Feigenbaum).

« Les principales composantes d'un système d'IA doivent être les connaissances, le raisonnement, la compréhension du langage naturel et l'apprentissage » (A. Turing).

« L'IA est l'étude des concepts qui permettent de rendre les machines intelligentes » (P. Winston).

« L'automatisation d'activités qui nous associons à la pensée humaine, comme la prise de décision, la résolution de problème ou l'apprentissage » (R. Bellman).

« L'IA est l'étude des facultés mentales à travers l'utilisation de modèles informatiques » (E. Charmiak et D. McDermott).

« L'IA est la science de programmer les ordinateurs pour qu'ils réalisent des tâches qui nécessitent de l'intelligence lorsqu'elles sont réalisées par des êtres humains » (M. Minsky).

« La question n'est pas de savoir si les machines peuvent avoir des émotions, la question est de savoir si elles peuvent être intelligentes sans émotion » (M. Minsky).

L'étude de comment programmer les ordinateurs pour qu'ils réalisent des tâches pour lesquelles les êtres humains sont actuellement meilleurs » (M. Rich et A. Knight).

« L'IA est la partie de l'informatique consacrée à l'automatisation de comportements intelligents » (G.L. Lugger et W.A. Stubbleeld).

« L'objectif de l'intelligence artificielle est, à long terme, de faire effectuer par un système informatique tout ce que peut faire l'homme en matière de raisonnement » (J.L. Laurière).

« L'IA commence là où l'informatique classique s'arrête : tout problème pour lequel il n'existe pas d'algorithme connu ou raisonnable permettant de le résoudre relève a priori de l'IA » (J.L. Laurière).

Citons maintenant une grande vérité avec Jacques Arsac : *« Après avoir longuement travaillé aux transformations de programmes, à simplifier au maximum le processus de création, je me trouve confronté à un des plus vieux problèmes de l'humanité : comment invente-t-on ? Il se trouve que la pratique de l'informatique m'a permis de trouver du neuf dans des sentiers infiniment battus. Y aurait-il donc dans l'approche informatique des problèmes une puissance créatrice nouvelle ? Je suis tenté de recommander*

des heuristiques en matière de création de programmes mais en même temps je crains toujours que l'habitude d'une certaine façon d'opérer ne bloque la possibilité de faire autrement dans des cas qui paraissent le nécessiter. Comme pour l'apprentissage du bridge, il y a des règles, et le débutant fait bien de s'y tenir, même s'il peut éventuellement en pâtir. Le bon joueur, cependant, est celui qui n'est pas esclave des règles ».

Et enfin : *« L'intelligence artificielle se définit comme le contraire de la bêtise naturelle »* (Woody Allen).

Dernier point capital… Les progrès de l'IA ont un point de passage obligé : la connaissance de l'intelligence humaine, or celle-ci est très difficile à cerner, tout ce qu'on peut dire est qu'elle se caractérise par l'adaptation à des situations nouvelles, l'apprentissage par l'usage (assimilation, accommodation), elle utilise le tâtonnement, l'analogie, elle reconnaît les « formes », elle utilise fortement le mimétisme, voire le conformisme d'où le succès d'une simulation de compréhension, et se dirige naturellement vers la synthèse.

Le problème central rencontré dans toutes les applications est celui de la connaissance et du raisonnement.

Historique

L'IA est née de recherches en démonstration automatique et en traduction, dès les années 1950. A cette époque, on ambitionnait sérieusement la traduction automatique, et le remplacement des experts humains par des systèmes experts possédant une base de connaissances évolutive.

Mais on s'est rendu compte assez vite que l'enthousiasme ne donnait pas les résultats escomptés, ce qui n'a pas empêché, dans les années 1970, de s'y abandonner de nouveau avec le langage Prolog. Cependant, on a compris la difficulté de la tâche. On ne fera des systèmes de compréhension du langage naturel que si on arrive à formaliser celui-ci, ce qui n'est toujours pas le cas. Car on ne sait pas bien comment le langage fonctionne. Et on ne pourra concevoir des systèmes intelligents que lorsqu'on comprendra comment fonctionne l'intelligence humaine. Là encore, on est très loin du compte. Mais le développement des systèmes-experts, le raisonnement par défaut, le traitement de l'imprécis et de l'incertain et l'apprentissage (connaissance acquise par généralisation à partir d'exemples) ont donné des applications parfois restreintes à des domaines particuliers, mais opérationnelles. Ainsi, Eliza *[Weizenbaum 1966]* est un célèbre programme destiné à leurrer son utilisateur par un dialogue comme

celui d'un patient avec son psychanalyste. Ce fut le début de réussite pour le fameux test de Turing. Ancêtre des robots conversationnels d'aujourd'hui, à l'époque, certains se sont fait prendre, mais, le sachant artificiel, il était assez facile de lui faire perdre pied. Chose de plus en plus ardue actuellement…

A partir des années 1990, après s'être concentrées sur des logiques non classiques, les recherches ont un peu ralenti ou se sont dirigées dans d'autres voies, les nombreuses applications industrielles de la logique floue et des réseaux de neurones sont apparues, l'évolution artificielle a pris son essor grâce à la plus grande capacité des machines et la fouille de données s'est beaucoup développée avec l'avènement d'internet.

Aujourd'hui, l'IA présente une diversité remarquable : reconnaissance vocale, des formes, nouveau rebond avec le traitement automatique du langage naturel, aide à la traduction, à la conception de vidéo, interfaces intelligentes, apprentissage automatique, fouille de données, fouille de textes ; diagnostic, systèmes experts, (médical, bourse, panne, etc.) ; jeux (échecs, dames, go, etc.) et partout où ont échoué des algorithmes classiques ; ordonnancement, satisfaction de contraintes ; optimisation sur des problèmes de parcours ou de charge ; robotique *[Ganascia1990]*.

L'intelligence naturelle

Sur ces mots vagues, beaucoup d'expressions mal définies nagent sans cesse dans l'immense océan des conversations et de la littérature. Descartes fut l'un des premiers a bien poser le problème. Rapidement, sans parler des interminables développements auxquels ils donnent lieu, en voici quelques-uns en rapport avec l'IA, les sciences cognitives, voire avec l'informatique théorique.

Parmi les intelligences les plus brillantes, Kurt Gödel 1906-1978, comme Copernic, fit faire un bond qualitatif à la connaissance.

Intelligence : une foule de livres et d'articles ont été écrits sur l'intelligence, ses définitions, ses rapports avec l'instinct, l'intuition, et les émotions, pour lesquelles personne n'est vraiment d'accord sur leurs définitions ou même leur dénombrement. Bien que mal définie, l'intelligence est mesurée par le QI mais qui ne note en fait, seulement un certain type d'intelligence…

Etymologiquement, *intelligentare* est la faculté de comprendre, et dans le cas le plus général, souvent résumée comme faculté de faire face à des situations nouvelles, donc d'adaptabilité et en cela constitue une notion s'opposant à l'instinct. L'instinct, lui, a un caractère plus automatique, inné et sans pensée structurée. Ceci étant, cette possibilité ainsi que celle du langage articulé est également innée. Le langage et ses corollaires, conscience et raisonnement, ne sont pas innés, mais leur potentialité, si.

Comprendre : du latin *« saisir avec »* donc éprouver la même chose, ce qui n'est pas tout à fait la même chose que « acquérir des explications » ou « posséder la connaissance ».

Conscience : on dit souvent que la conscience de soi et l'interrogation sur la mort est la ligne de démarcation entre l'homme et les hominidés antérieurs. La conscience véritable n'a pu apparaître qu'avec le langage articulé, lequel s'apprend chez les enfants par mimétisme, essais – erreurs – rectifications et donc conformisme.

Le raisonnement de sens commun

On nomme ainsi le raisonnement humain tel qu'on l'entend dans les conversations ou les écrits divers. Ce raisonnement utilise plusieurs modes que l'on va illustrer. En premier lieu, il fonctionne beaucoup par une logique de défauts, ainsi prenons un exemple simpliste :

Les Suédois sont blonds,

x est suédois et les parents de x sont espagnols $\rightarrow$ *x est brun,*

x est suédois et les parents de x sont espagnols et blonds $\rightarrow$ *x est blond*

Ces trois règles vont du général au particulier, une règle générale est tenue pour vraie tant qu'une règle plus spécifique ne s'applique pas. Et une règle peut être dite plus spécifique tout simplement si elle a plus de conditions, c'est-à-dire un plus grand nombre de prémisses. Le raisonnement de sens commun fonctionne aussi beaucoup par analogies :

S'il pleut, les oiseaux ne chantent pas, alors s'il neige, sans doute aussi.

Il met aussi très souvent en jeu des règles graduelles :

Plus le climat est chaud, plus la cuisine est pimentée.

Cela se fait par déduction, mais aussi par de fréquentes abductions :

x est très blond, il doit être scandinave.

Mais le raisonnement de sens commun confond souvent, pour ne pas dire toujours, les règles floues (vraies à un certain degré) et les prédicats flous qu'il a tendance à rendre binaires. Dans l'exemple « les Suédois sont grands », on peut considérer cette règle statistiquement vraie à un degré 0,8 mais « grand » est lui-même un prédicat plus ou moins vrai. Il serait très simpliste de dire que « grand » est défini au seuil de 1m80, c'est

typiquement un prédicat flou comme « blond ». Il y a donc deux degrés qui interviennent dans la confiance que l'on peut accorder à chaque utilisation de ce genre de règle.

Le raisonnement de sens commun apprend enfin par induction, c'est-à-dire en forgeant ses propres règles à partir de son expérience, mais il apprend aussi par échanges avec ses congénères, ce que commencent à faire les systèmes multi-agents, simples mais communicants.

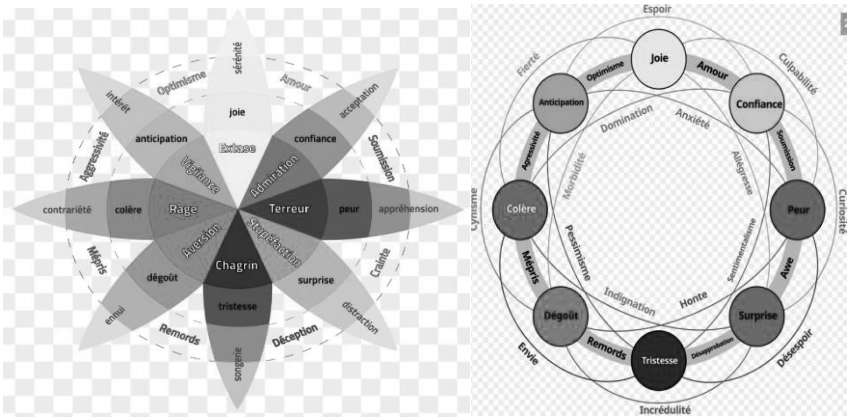
Reste que l'intelligence est toujours collective, d'abord l'association de cellules formant des êtres multicellulaires, mais ensuite les associations telles que lichens, coraux, et toutes les symbioses avec d'autres pour résoudre, voire effectuer une fonction, par exemple, la digestion de l'homme qui abrite dix fois plus de bactéries que ses propres cellules. A chaque fois, qu'il y a un problème (non formalisé) à résoudre… Cette intelligence partagée s'est faite par évolution tout comme les organisations d'insectes sociaux. Quant à l'intelligence proprement dite humaine, sans les parents et tout l'environnement social, elle ne se développe guère, ceci étant, elle se développe par renforcement (punitions-récompenses) et en étant liée de façon indissociable aux émotions.

Sensations : effet des sens, mais l'intelligence, l'intuition n'est-elle pas un autre sens, c'est-à-dire un moyen de se faire des idées, donc un sixième sens pour la conceptualisation ?

Emotions : réponse automatique du corps (réflexe) à une situation : *surprise, plaisir-joie, peur, colère, confiance-amour, tristesse, désir, aversion-dégout, honte*… Elles sont dénombrées entre 5 et 17 dans certains systèmes de robotique. Avec toute une *graduation* dans toutes les langues, par exemple *approbation – confiance – gentillesse – affinité – amitié – engouement – amour – adoration – dévotion* ou encore *honte – regrets – remords*. Sur l'instantanéité aussi, on peut dire : **pulsion** = premier instant de l'émotion et **sentiment** = émotion durable et aussi **indifférence** = absence d'émotion. On tente de définir des émotions de base (6 pour Descartes, 8 pour l'octogone ci-dessous, voire bien plus), 4 universellement reconnues par les expressions faciales : peur, tristesse, colère, plaisir, puis des émotions composées par exemple : *surprise* + *peur* + *tristesse* = *colère*.

L'IA a commencé à s'intéresser aux émotions avant 2000, surtout par les expressions faciales en traitement d'images, et par d'autres traits physiologiques captés comme la sudation et le rythme cardiaque. Le but étant d'établir une relation entre ces signaux physiologiques et l'état émotionnel. L'**intelligence émotionnelle** serait plutôt la capacité d'accéder et de contrôler ses émotions et celles des autres. Le psychologue Plutchik est celui qui a le plus essayé de caractériser 8 émotions de base (et les 4 fondamentales joie, peur, tristesse, colère et leurs degrés). Soulignant que

l'évolution naturelle a développé ces réactions dans un but de survivance à comparer à nos cousins mammifères et autres. En IA, la reconnaissance émotionnelle, bien difficile, progresse néanmoins.

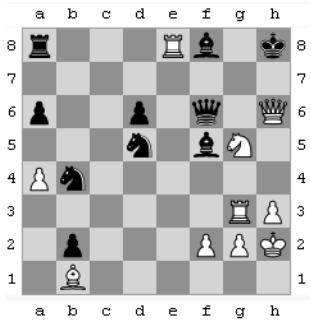


La roue de Plutchik et ses descriptions de comportements comme combinaison de paires les définissant en sentiments.

Remarque sur les jeux

Certains jeux sont parfaitement résolus. C'est le cas de Puissance 4 ou le jeu d'Othello où on a montré qu'il existait une stratégie gagnante pour le joueur qui commence à jouer *[Alliot, Dubois 1995]*. Cette stratégie étant stockée dans une base de données, il est facile pour un joueur artificiel de suivre cette stratégie et de gagner systématiquement.

Pour les dames, un programme Chinook basé sur l'algorithme alpha-béta (chapitre 1) est devenu champion du monde en 1994. Ce joueur utilise également une bibliothèque de fins de partie (pour tous les damiers comportant 8 pièces ou moins).



Concernant les échecs, le fameux DeepBlue bat Kasparov en 1997, grâce à un programme écrit en C à l'université Carnegie-Mellon. Depuis, toutes les confrontations tournent systématiquement à l'avantage de la machine grâce à l'utilisation de l'algorithme alpha-béta, chaque nœud de l'arbre possédant environ 40 possibilités filles ; un bon joueur sélectionne très vite une demi-douzaine de coups importants sans analyser ceux qui lui paraissent moins rentables *[De Groot 1965]*.

Le jeu de Go, resté longtemps coriace à l'algorithmique, a fini par être dompté. Les joueurs artificiels étaient de niveau amateur. L'algorithme alpha-bêta est peu utilisable dans ce cas (un coup possède environ 360 réponses). La prime promise pour le premier programme qui bat un champion humain a été réalisée pour Alphago en 2016 (puis alphazero, s'adaptant lui-même) et a battu le meilleur joueur Ke Jie en 2017. Avec des méthodes évolutionnaires simple comme « Monte-Carlo » qui consiste à prendre une succession d'ensembles importants de coups à jouer pour sélectionner les meilleurs et à faire de l'auto-apprentissage.

Les joueurs humains utilisent des configurations spatiales qui ressemblent à des parties connues. Le sens commun utilise donc beaucoup l'analogie fondée sur des notions de similarités restant souvent bien difficiles à formaliser.

L'auto-organisation

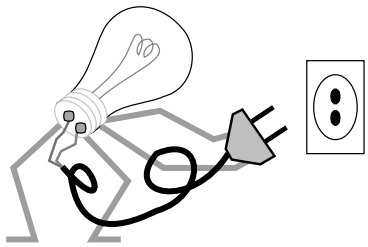
A propos du « bootstrap » ou l'émergence d'une organisation à partir de « rien » ou de « peu » on lira les papiers de *[Dormoy 1993]* et ceux de *[Pitrat 1995]* qui développa de gros systèmes de résolution de problèmes divers en essayant de programmer plusieurs niveaux de connaissances, méta-connaissances, méta-méta-connaissances…

On note toutefois les intéressantes idées de *[Laurière 1987]* : Le rêve plus ou moins avoué est de faire résoudre une grande variété de problèmes à une machine intelligente universelle, à savoir un système capable de résoudre n'importe quel problème sans qu'on lui donne beaucoup de précision dans les données. C'est sans doute une chimère, mais le domaine de l'évolution artificielle va un peu dans ce sens car en donnant un problème avec ses contraintes à optimiser, l'expérience montre qu'à partir d'une population aléatoire de « solutions potentielles » à ce problème, il s'en détachent toujours des « individus » meilleurs et donc des solutions approchées. Mieux que cela, il est des solutions surprenantes, qui optimisent la fonction avec les contraintes que l'on a données et qui conduisent à s'interroger sur son bien-fondé. Le méta-problème est donc de bien formaliser le problème à résoudre. D'autre part, les algorithmes imaginés en IA ont des paramètres à fixer au départ, comment les apprécier ? On en est souvent venu à des méthodes hybrides mêlant plusieurs algorithmes, mais plus encore à vouloir fixer les paramètres d'un algorithme par un autre algorithme, qui lui-même a besoin de connaissances initiales.

Par ailleurs, *[Wolpert, Macready 1997]* avec le « No free lunch theorem » établissent en termes probabilistes qu'il ne peut y avoir d'algorithme d'optimisation universel. L'auto-organisation, le « bootstrap », c'est la quête du Graal en IA. Mais s'en approche, par exemple,

qu'un robot japonais capable de « penser de lui-même » afin d'apprendre à résoudre des problèmes auxquels il n'a encore jamais été confronté.

Cette capacité est d'ailleurs une des meilleures définitions de l'intelligence naturelle. Résoudre des problèmes mal définis, prendre des initiatives pour aller chercher des méthodes et des connaissances dans le web, apprendre, se modifier, communiquer avec les humains, tels sont les buts eux-mêmes assez vagues et mal définis de l'IA, donc un apprentissage non supervisé.



Ci-dessus, un des objectifs de l'IA

Références

Alliot J.M. Dubois N. *A genetic algorithm to improve an othello program*, Artificial Evolution, Lecture Notes in Computing Science 1063, p307-319, 1995

Bouzy B. *Un modèle à objets pour décrire les groupes de jeu de Go,* Laforia 94/17, 1994

Chaitin G., *Hasard et complexité en mathématiques*, Flammarion, 2005

Crevier D. *Une histoire de l'intelligence artificielle*, Seuil, 1999

De Groot A.D. *Thought and choice in chess*, Ed. Mouton, 1965

Descartes R. *Le discours de la méthode*, (1637) Editions sociales, 1950

Descartes R. *Règles pour la direction de l'esprit* Gallimard 2016

Dormoy J.L. *Où aller en IA ?* Sur *dormoy.org/JLuc/Papers/Bootstrap93.pdf,* 1993

Ganascia J.G., *L'âme-machine: les enjeux de l'intelligence artificielle*, Seuil, 1990

Ganascia J.G. *Les sciences cognitives* Le Pommier 2006

Gross M. *The Construction of Local Grammars.* The MIT Press, 1997

Laurière J.L., *Intelligence artificielle,* Eyrolles, 1987

Laurini R. *Cours sur les systèmes d'informations géographiques* INSA – Univ. Lyon, 2011

Le Moigne J.L. *La modélisation des systèmes complexes,* Dunod, 1999

Morin E. *L'intelligence de la complexité,* L'Harmattan, 1999

Pitrat J. *De la machine à l'intelligence,* Hermès, 1995

Plutchik R. Hope R. Conte H.R. *Circumplex Models of Personality and Emotions*, 1997

Poincarré J.H. *La science et l'hypothèse,* Flammarion, 1968

Russel S. Norvig P. *Intelligence artificielle, une approche moderne,* Pearson, 1995, 2021

Sabah G. *Vers une technologie de la conscience ?* Présentation en assemblée plénière de l'Académie des technologies, 2012

Sowa J.F. *Conceptual Structures* Addison-Wesley, 1984

Simon H.A. *Can there be a science of complex systems ?* Proc. of Int. Conf. of complex systems Cambridge, 1997

Torré F., http://www.grappa.univ-lille3.fr/~torre/Enseignement/Bibliographies/ia.php

Weizenbaum J., *Eliza, a computer program for the study of natural language communication between man and machine,* ACM, 1966

Wolpert D. Macready W. *No free lunch theorem for optimization,* IEEE Transactions on Evolutionary Computation, 1 : 67, 1997

CHAPITRE 1

PARCOURS D'ARBRES ET DE GRAPHES

Ce premier chapitre résume un peu les premières interrogations des débuts de l'intelligence artificielle. Comment planifier une action, comment explorer les actions possibles à partir d'un état du problème considéré ? Beaucoup de problèmes se ramènent à chercher un chemin dans un graphe, mais avant cela, il aura fallu définir les nœuds de ce graphe. C'est tout le problème de la représentation des états d'un problème et plus généralement de la représentation des connaissances.

Il n'est jamais facile a priori de dire si une modélisation est meilleure qu'une autre. L'espace de recherche est plus ou moins vaste et les considérations informatiques de programmation ont leur influence sur cette représentation souvent plus encore que les considérations algorithmiques de recherche d'une solution. Poser formellement un problème, c'est déjà la moitié du travail en vue de sa résolution.

1.1 SPECIFICATION ET PLANIFICATION

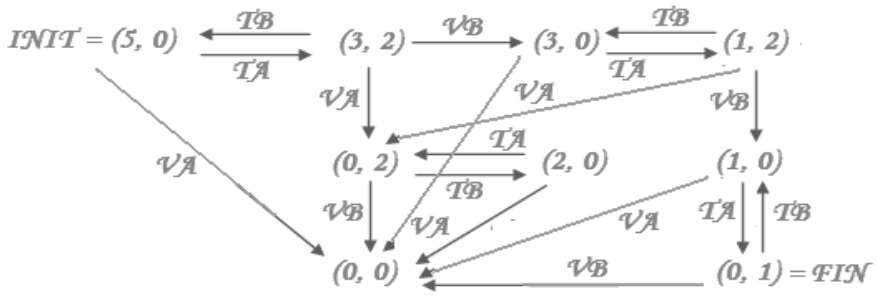
Un premier exemple montrera, malgré sa simplicité, combien l'esprit humain se précipite sur la résolution d'un problème avant de le spécifier calmement et de songer à une représentation simple des états que peut prendre un système simple soumis à des opérations simples.

1.1.1 Définition d'un espace d'états : le problème des cruches

Etant donnés deux cruches, A de 5 litres initialement pleine et B de 2 litres initialement vide, on cherche à obtenir A vide et exactement un litre dans B. Les seules opérations possibles sont : vider une cruche (dans l'évier), transvaser le contenu d'une cruche dans l'autre complètement si c'est possible sans faire déborder ou jusqu'à ce que la cruche réceptrice soit pleine *[Foutelet]*.

L'exercice consiste à ne pas « chercher à résoudre le problème » mais à en donner une représentation des états du problème puis définir les opérateurs, enfin dessiner le graphe complet de tous les états possibles.

> La représentation la plus simple semble être le couple (contenu de A, contenu de B)
> Les opérations Vider et Transvaser peuvent s'écrire ainsi ou bien avec 3 arguments en donnant le nom de la cruche.
> VA (x, y) $\rightarrow$ (0, y) TA (x, y) $\rightarrow$ (max(0, x – 2 + y), min(2, x + y)
> VB (x, y) $\rightarrow$ (x, 0) TB (x, y) $\rightarrow$ (min(5, x + y), max(0, y – 5 + x))



1.1.2 Exemple de la pizza

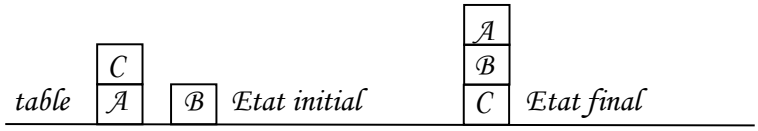
Décrire l'achat d'une pizza par téléphone.

> On peut simplement décrire les faits T = avoir un téléphone qui fonctionne, A = avoir de l'argent, C = passer commande, M = être à la maison, R = être au restaurant, P = avoir la pizza et les actions appeler, aller, commander, acheter. Les états initial et final sont alors Init = {T, A, M} et Fin = {M, P}.

1.1.3 Exemple des cubes

Trouver les prédicats nécessaires à l'écriture des conditions pour passer de l'état initial (à gauche) à l'état final (à droite). Il s'agit encore d'un problème de planification comme celui du singe devant déplacer un tabouret pour chercher une banane...

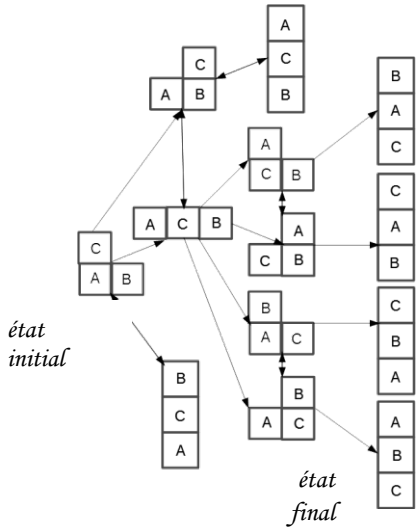
Les seuls opérateurs sont « prendre un cube », le « poser sur la table » ou « sur un autre cube ».



Les prédicats peuvent être libre(X), surtable(X), enmain(X), mainvide, sur(X, Y).

> Mais on peut toujours trouver autre chose.
> Ces états sont Init = {libre(c), libre(b), surtable(a), surtable(b), sur(a, c), mainvide}
> Final = {libre(a), sur(a, b), sur(b, c), mainvide, surtable(c)}
> Les opérateurs (en notant les présupposés, suppressions et ajouts) :
> poser(X) pré et suppr : enmain(X), ajout : surtable(X), libre(X), mainvide
> soulever(X) pré et suppr : mainvide, surtable(X), ajout : enmain(X)
> empiler(X, Y)
> pré et suppr : enmain(X), libre(Y),
> ajout : mainvide, sur(X, Y)
> depiler(X, Y) pré et suppr : mainvide, libre(X), ajout : enmain(X), libre(Y)
> Ici la solution est simple :
> depiler(c, a), poser(c), empiler(b, c), soulever(a), empiler(a, b)

Graphe complet ci-après (il n'y a que 3 emplacements sur la table) :



Bien sûr, les règles peuvent être multiples, ordonnées dans le temps, avoir des pondérations, on pourrait décrire des plans conditionnels etc.

ALGORITHME DE PLANIFICATION

Les premiers algorithmes de planification sont *GPS* (*General Problem Solver, E. Newell et H. Simon 1961)*, STRIPS (*STanford Research Institute Problem Solver, Richard Fikes et Nils Nilsson en 1971),* GraphPlan…

Après la définition des prédicats et des opérateurs (à chaque opérateur est associé une liste de préconditions, une liste de suppression et une liste d'ajouts correspondant à son action), les états du problème sont empilés et on peut décrire de façon informelle :

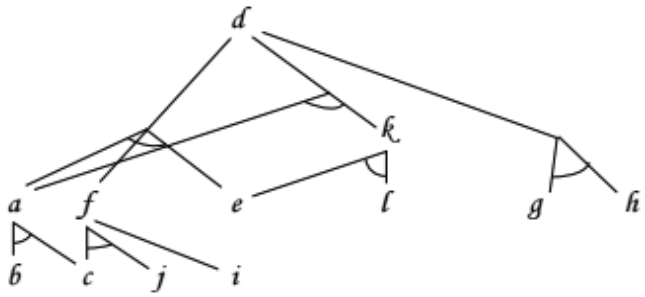
satisfaction (Buts, e, pile) (* e état courant *)
pour chaque b ∈ Buts faire
    NouvelEtat e' ← realisation (b, e, pile)
    si e' = échec alors échec
    sinon si tous les b ∈ Buts sont satisfaits dans l'état e' alors e'
    sinon échec

réalisation (b, e, Pile)
    si b satisfait dans l'état e alors e
    sinon si b ∈ pile alors échec
    sinon pour chaque operateur op pouvant satisfaire le but faire
        e' ← appliquer (op, e, b :: pile) (* on rajoute le but en question *)
        si e' ≠ échec alors b satisfait dans l'etat e et retourner e'
        sinon échec

appliquer (op, e, Pile)
e' ← satisfaction (préconditions de op, e, pile)
si e' <> échec alors faire l'action de op,
e' ← e' – liste de suppressions de op
et retourner e' :: liste des ajouts de op
sinon échec

1.1.4 Exemple d'hypergraphe

Il s'agit simplement d'est un graphe qualifié de *et* / *ou* dans lequel les branches reliées par un arc signalent une conjonction *et*. En partant du but d, les règles sont :
b, c → a; a, e, f → d; a, k → d; i → f; c, j → f; g, h → d; e, l → k.



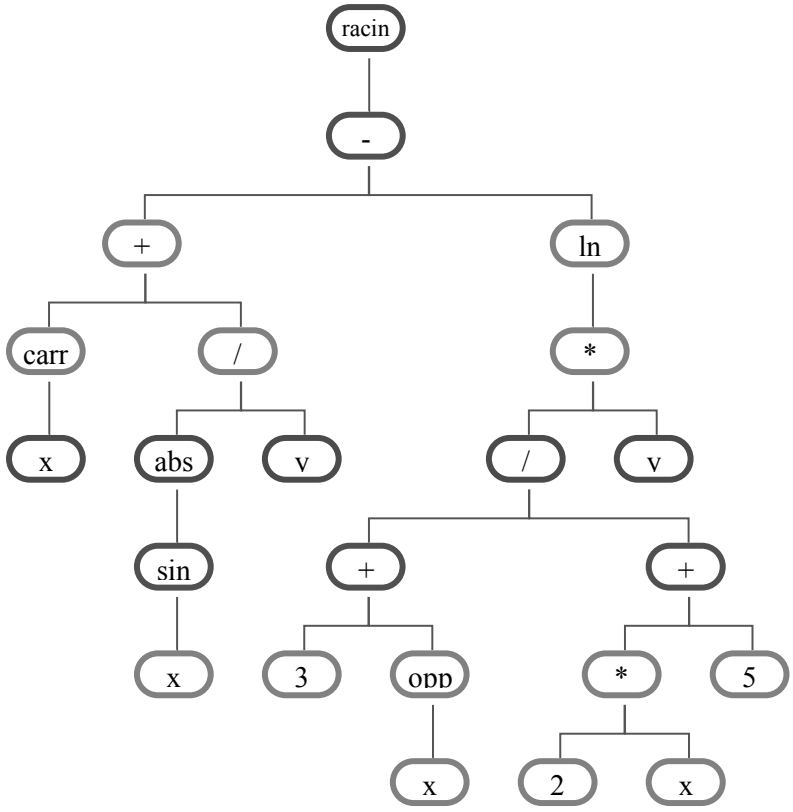
STRATÉGIES DE PARCOURS D'ARBRE

Le but final de tout problème, se traduisant par un graphe d'états, est d'obtenir une solution, mais aussi la suite des actions pour y parvenir, voire l'ensemble des solutions, d'où l'importance des méthodes de parcours de graphes en recherche opérationnelle et d'abord pour les arbres. Rappelons les stratégies de lecture d'arbres. Si un arbre est (récursivement) la donnée (r, f) d'une racine et de la liste de ses sous-arbres fils, alors sa lecture peut être réalisée par des parcours différents.

Le plus simple est de prendre par exemple une expression mathématique *E* ici écrite suivant la façon traditionnelle, ou notation infixe, ce qui signifie par exemple que le signe + se trouve entre les deux termes de l'addition.

$$E=\sqrt{x^2+\frac{|\sin(x)|}{y}-\ln y\,\frac{3+(-x)}{2x+5}}$$

En fait, une telle expression correspond à l'arborescence suivante, et la notation infixe résulte d'une lecture gauche - racine - droite, de l'arbre.



La notation suffixée ou polonaise, (utilisée dans le langage de programmation Forth), résulte d'une lecture gauche - droite - racine, ce qui donne par exemple ici :

$$E_S = x\ carré\ x\ sin\ abs\ y\ /\ +\ 3\ x\ opp\ +\ 2\ x\ *\ 5\ +\ /\ y\ *\ Ln\ -\ \sqrt{}$$

On a noté par le signe – la soustraction de deux arguments et *opp* la fonction *opposé* à un seul argument. Si le même symbole n'est jamais utilisé pour des opérateurs d'arités différentes, alors les parenthèses deviennent inutiles. Cette notation est, de loin, la plus rationnelle, car elle établit une correspondance entre l'ordre d'écriture de gauche à droite et l'ordre opératoire, ceci permettant de débuter les calculs avant même que l'expression soit terminée d'être donnée.

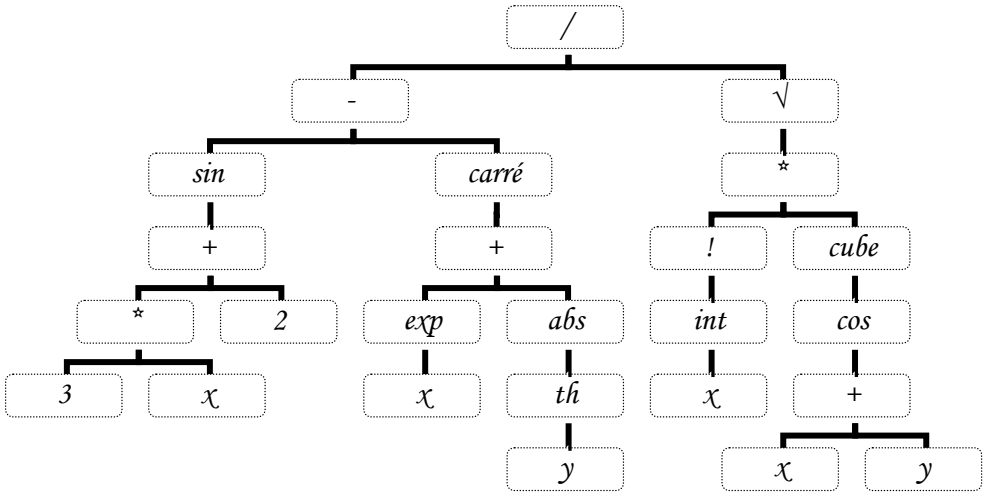
Par contre, c'est la notation préfixée (correspondant à la lecture racine-gauche-droite de l'arbre) du Lisp, qui consiste à écrire au contraire, la fonction avant ses opérandes, ce qui donne ici entièrement parenthésée :

$$E_p = (\sqrt{}\ (-\ (+\ (carré\ x)\ (/\ (abs\ (sin\ x))\ y))\ (Ln\ (*\ (/\ (+\ 3\ (opp\ x))\ (+\ (*\ 2\ x)\ 5))\ y))))$$

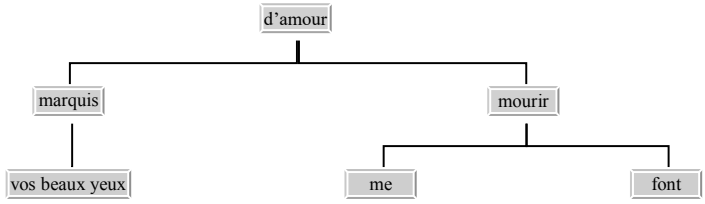
1.1.5 Tracer l'arbre de l'expression algébrique

$$E = \frac{\sin(3x+2) - (e^{x} + |\text{th y}|)^{2}}{\sqrt{(\text{Int } x)! \cdot \cos^{3}(x+y)}}$$

En déduire cette expression par les parcours gauche-droite-racine (profondeur d'abord, suffixée ou polonaise) et racine-gauche-droite, (expression préfixée). On note *int* la partie entière.



Es = 3 x * 2 + x exp y th abs + carré + x int fac x y + cos cube * rac /
Ep = / – sin + * 3 x 2 carré + exp x abs th y rac * fac int x cube cos + x y
Lire l'arbre ci-contre en rgd, grd et gdr.



L^{suffixe} = vos beaux yeux, marquise, me font mourir d'amour
$L^{\text{préfixe}}$ = d'amour, marquise, vos beaux yeux, mourir, me font
L^{infixe} = marquise, vos beaux yeux, d'amour, me mourir, font

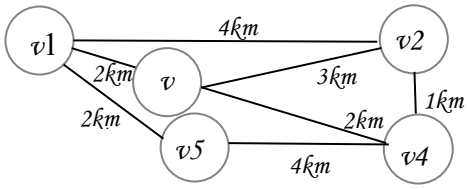
1.1.6 Transformation en postfixe d'une expression parenthésée usuelle

Par exemple, pour *E = sin(3x + 2) – exp(5*x / 3)* il faudra obtenir l'expression postfixée *3 x * 2 + sin 5 x * 3 / exp –*.

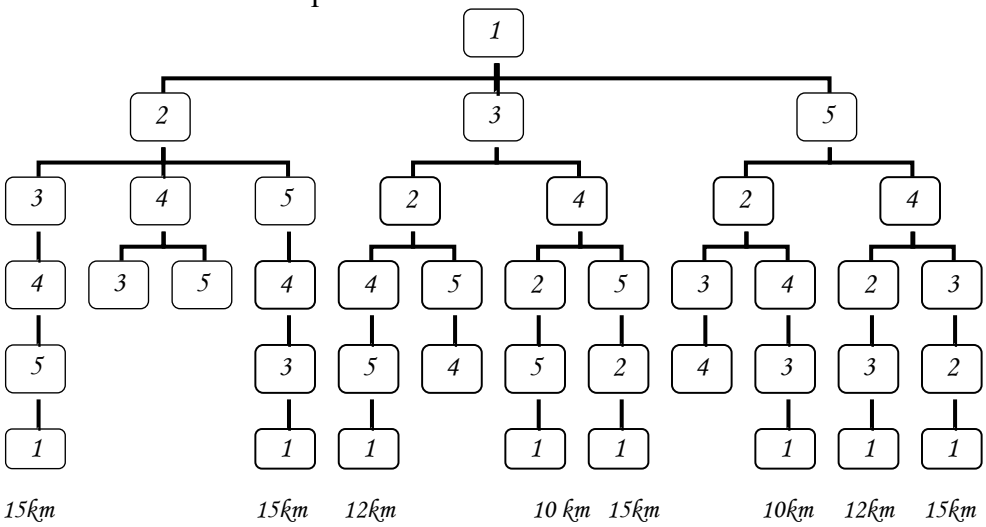
```
Solution en Lisp, d'abord tout parenthèser :
(defun postfix (E) (cond ((atom E) E)
        ((null (cddr E)) (list (postfix (cadr E)) (car E)))
        ((list (postfix (car E)) (postfix (caddr E)) (cadr E)))))
Exemple (postfix '((sin ((3 * x) + 2)) – (exp ((5 * x) / 3))))
→ ((((3 X *) 2 +) SIN) (((5 X *) 3 /) EXP) –)
```

1.1.7 Exemple de parcours systématique pour le voyageur de commerce

En partant de la ville *v1*, construire l'arbre de toutes les possibilités de réalisation d'un parcours complet avec retour en *v1*, sans passer deux fois par la même ville (chemin hamiltonien).



L'arbre est donné par :



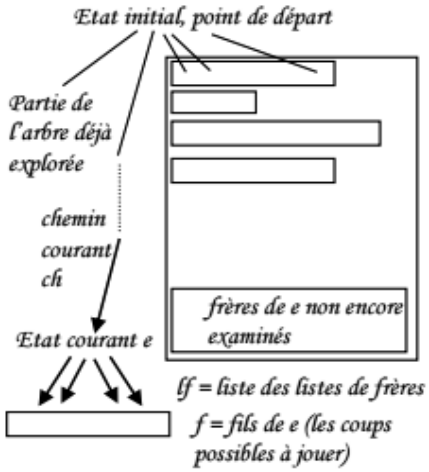
Arbre complet des boucles possibles de la ville 1 et retour en 1 avec totaux, on voit donc qu'il y a deux plus courts chemins de 12km.

1.2 L'ALGORITHME DU « BACKTRACKING »

Cet algorithme, appelé en français « essais avec retour en arrière », consiste à lire un arbre suivant la stratégie « racine - gauche - droite ». Ainsi, on retient en mémoire à chaque instant, une branche de l'arbre avec à chacun de ses étages, la liste des fils non encore explorés. De cette façon, on ira beaucoup plus loin dans la résolution informatique d'un problème qu'en développant entièrement toutes les branches jusqu'à une profondeur donnée. Cette dernière méthode, dite de lecture en largeur, ne tient pas longtemps avant d'arriver au « stack overflow ».

Chercher à résoudre un problème en construisant pas à pas l'éventuelle solution, quitte à revenir en arrière dans la suite des choix à faire, est connu sous le nom d'algorithme à retour ou « backtracking ». Il s'agit en fait d'explorer une arborescence en suivant l'ordre racine-gauche-droite encore appelé « parcours en profondeur d'abord ». L'exemple le mieux connu d'un tel problème est celui des reines de Gauss : placer 8 dames sans qu'aucune prise ne soit possible, sur un damier 8*8.

Dans le schéma ci-contre, on considère qu'à chaque état *e* du problème, on a une suite d'états *ch* qui y a amené et que *f* est la liste des états suivants.
Le cas où la liste *f* est vide, se décompose en deux, si rien n'a pu être choisi car *ch* est vide, c'est l'échec, sinon on retire le dernier choix de *ch*, et on remonte à l'examen des frères de l'état précédent, c'est le retour en arrière : une remontée dans l'arbre. On tiendra donc en réserve une liste *lf* des listes de frères non encore explorés.



Le troisième cas indique que si le premier des choix possibles arrive au but du problème, alors on renvoie cette solution, qui est la première rencontrée. Le cas suivant exprime une descente en ajoutant l'état suivant à la liste *ch*. Sinon, on va voir les frères, c'est un déplacement horizontal à droite dans l'arbre.

L'algorithme général du backtracking est un peu ce qu'il y a de plus délicat en programmation. Il peut se traduire, en gros, par une fonction

s'exprimant en 5 cas, et renvoyant le chemin qui aboutit au premier état solution.

On peut écrire des fonctions *but* et *valide* indiquant une valeur booléenne associée à un état *e*, et la fonction *fils* renvoyant la liste des états suivants d'un état *e*.

L'état courant *e* constituant la tête de *ch*.

Bien entendu, ces fonctions sont à écrire suivant le problème considéré, la fonction générale, dans les 5 cas cités, est alors :

```
fonction bak(f, ch, lf) = (* renvoie le chemin aboutissant au premier état solution *)
        si f = vide alors si ch = vide alors vide ECHEC
        sinon bak(tete(lf), queue(ch), queue(lf)) REMONTEE
        sinon si but (tete (f)) alors tete(f) suivi de ch SUCCES
        sinon si valide(tete(f)) alors bak(fils(tete(f)), tete(f) suivi de ch,
                                        queue(f) suivi de lf) DESCENTE
        sinon bak(queue(f), ch, lf) PASSAGE HORIZONTAL A DROITE
```

Le mieux est de le faire à la main pour le problème des dames de Gauss sur un échiquier de 4 sur 4.

1.2.1 Décomposition d'un entier *s* grâce à une liste d'entiers.

Dans ce jeu du « compte est bon », il faut par exemple que la décomposition de 7 avec les valeurs (3 2 1 4 3) renvoie comme première solution (3 1 3).

En Lisp avec l'appel initial (decomp s () lr ())

```
(defun decomp (s ch lr lf) (cond
; ch = liste des choix, lr = liste des nombres restants, lf = liste des listes de frères
        ((eq s 0) ch)
        ; le paramètre s est compté négativement, on stoppe avec succès si s = 0
        ((null lr) (if (null ch) 'impossible
                (decomp (+ (car ch) s) (cdr ch) (car lf) (cdr lf)) )) ;  remontée
        ((< s (car lr)) (decomp s ch (cdr lr) lf))
        ; le premier nombre est trop grand, voir à droite
        (t (decomp (– s (car lr)) (cons (car lr) ch) (cdr lr) (cons (cdr lr) lf)) )))
        ; descente
```

En Caml, la fonction de paramètres l'entier *s* et une liste d'entiers *ln*, donnant la première solution liste d'entiers pris parmi *ln* dont la somme est *s*, peut s'écrire grâce à une exception nommée « Impossible ».

```
exception Impossible;;
let rec decomp s ln = match ln with
                [] -> if s = 0 then [] else raise Impossible
                | x::q -> try x :: (decomp (s – x) q) with Impossible -> decomp s q;;
decomp 13 [3; 4; 3; 1; 8; 5; 6; 2];; → [3; 4; 3; 1; 2]
```

Maintenant si on veut toutes les solutions, la fonction donnant tous les résultats :

```
let rec tdec s = function [] -> []
        | (x :: q) -> if x > s then tdec s q
        else (tdec s q) @
                (if s = x then [[s]] else map (fun t -> x :: t) (tdec (s – x) q));;
```

A noter qu'il s'agit d'une concatenation avec x rajouté à toutes les solutions sans x.

```
tdec 7 [3; 2; 1; 4; 3];; → [[4; 3]; [2; 1; 4]; [3; 4]; [3; 1; 3]]
```

Ou encore avec deux fonctions mutuellement récursives, *ttdec* a pour arguments la liste « *sol* » des listes solutions déjà trouvées, la liste de chiffres choisis *lc*, la somme à annuler *s* et la liste de chiffres restants.

```
let rec tdec s = function [] -> []
        | (x :: q) -> if x > s then tdec s q else ttdec (tdec s q) [x] (s – x) q
and ttdec sol lc s r = if s = 0 then lc :: sol
else (match r with [] -> sol
        | (x :: q) -> ttdec (sol @ (ttdec [] lc s q)) (x :: lc) (s – x) q);;

tdec 7 [3; 1; 4;2; 3];; → [[2; 4; 1]; [3; 4]; [4; 3]; [3; 1; 3]]
tdec 7 [3; 2; 1; 4; 3];; → [[4; 1; 2]; [3; 4]; [4; 3]; [3; 1; 3]]
tdec 17 [3; 6; 5; 1; 4; 2; 3];; → [[6; 5; 4; 2]; [6; 5; 1; 2; 3]; [3; 5; 4; 2; 3]; [3; 6; 1; 4;
3];  [3; 6; 5; 3]; [3; 6; 5; 1; 2]]
```

PROBLÈME GÉNÉRAL

Dans une fonction de parcours très générale nommée *bak*, l'argument *fils* doit être une fonction qui à un état du problème associe la liste de ses fils (les coups suivants possibles dans un jeu), *init* est l'état initial ou racine, et *but* est la fonction booléenne indiquant si un état (un nœud de l'arbre) est terminal. En général, pour tous ces problèmes, c'est la fonction *fils* qui est la plus difficile à écrire. Dans ce qui suit, *ch* est le chemin complet exprimé à l'envers, c'est-à-dire la liste des nœuds parcourus de la racine à l'état courant *e*, enfin *r* désigne la liste des états restant à explorer.

En Caml light, la fonction générale *bak* a pour arguments la fonction calculant les fils d'un nœud (de type a'), celle indiquant le but atteint et l'état initial, elle ne fait que lancer la fonction auxiliaire *bak'*. (*rev, mem et map* ont des noms variables suivant les dialectes camllight, Ocaml, wincaml)

```
let bak fils but init =
        let rec bak' ch reste = match reste with [] -> false, []
                                        | (e :: r) -> (if but e then true, rev (e :: ch)
                                                else if mem e ch then bak' ch r
                                                else bak' (e :: ch) ((fils e) @ r))
        in bak' [] [init] ;;

val bak : ('a -> 'a list) -> ('a -> bool) -> 'a -> bool * 'a list = <fun>
```

1.2.2 Application de cette fonction générale au problème du « compte-est-bon ».

On définira le retrait d'un élément d'une liste, la différence ensembliste, les fils suivants d'un état du problème et la fonction *but*.

```
let rec ret x a = match a with [] -> [] | (y :: q) -> if x=y then q else y :: (ret x q);;
let rec dif a b = match b with [] -> a | (x :: q) -> dif (ret x a) q;;
let suivants ln e = List.map (fun x -> x::e) (dif ln e);;
let decomp s ln = bak (suivants ln) (fun e -> (s = List.fold_left (+) 0 e)) [] ;;

suivants [1;2;3;4;5;3;8;1] [2;3;1];;
→ [[4; 2; 3; 1]; [5; 2; 3; 1]; [3; 2; 3; 1]; [8; 2; 3; 1]; [1; 2; 3; 1]]
decomp 7 [3; 2; 1; 4; 3];; → (true,  [[]; [3]; [2; 3]; [1; 2; 3]; [4; 1; 2; 3]; [3; 4; 1; 2;
3]; [3; 1; 2; 3];   [4; 3; 1; 2; 3]; [4; 2; 3]; [1; 4; 2; 3]; [3; 1; 4; 2; 3]; [3; 4; 2; 3]; [1; 3;
4; 2; 3]; [3; 2; 3]; [1; 3; 2; 3]; [4; 1; 3; 2; 3]; [4; 3; 2; 3]; [1; 4; 3; 2; 3]; [1; 3]; [2; 1;
3]; [4; 2; 1; 3]; [3; 4; 2; 1; 3]; [3; 2; 1; 3]; [4; 3; 2; 1; 3]; [4; 1; 3]; [2; 4; 1; 3]; [3; 2;
4; 1; 3];   [3; 4; 1; 3]; [2; 3; 4; 1; 3]; [3; 1; 3]])
C'est tout le chemin arrivant à la première solution
```

POUR UN ASPECT PLUS GÉNÉRAL, CHANGEONS DE LANGAGE AVEC HASKELL

Avec une syntaxe très transparente, (page 287, annexe 12) Haskell est fonctionnel, avec calcul automatique des types, classes et passage paresseux des paramètres ce qui permet quelques prouesses), il est facile de chercher le problème du « compte est bon », des tours de Hanoï ou des reines de Gauss. Ici, on a une fonction extrêmement courte (le deux-points : est le *cons* du Lisp, équivalent au :: du Caml) :

```
bak fils but init = bak' [init] [] where
                    bak' [] ch = (False, ch)
                    bak' (x : q) ch = if but x then (True, x : ch)
                                      else if elem x ch then bak' q ch
                                      else bak' ((fils x) ++ q) (x : ch)
```

Pour le compte est bon avec les valeurs contenues dans une liste *ln*, on va représenter chaque état intermédiaire du problème par le couple formé par la liste *lc* des nombres choisis et celle *lr* de ceux restant disponibles, la difficulté est d'écrire le but à atteindre, surtout la liste des *fils* ou états suivants d'un état, et enfin une fonction de visualisation *vue*. Noter ici, deux fonctions anonymes, qui ne sont définies que lorsqu'elles sont passées.

```
compte s ln = vue (bak (\ (lc, lr) -> map (\x -> (x : lc, ret x lr)) lr)
                                      (\ (lc, _) -> sum lc == s) ([], ln))
                    where vue (b, le) = if b then fst (head le) else []
```

En prenant des définitions *ret* et *dif* pour supprimer un élément d'une liste et pour la différence ensembliste.

```
dif l [] = l
dif l (x:m) = if elem x l then dif (ret x l) m else dif l m
ret x [] = []
ret x (y : q) = if x == y then q else y : (ret x q)
```

Exemples : compte 7 [3,2,1,4,3] → [3,1,3] compte 7 [3,2,1] → []

On peut aussi avoir une écriture ensembliste très simple :

```
suiv e = [i::e | i <- [1..n], valide i e]
```

Par ailleurs, pour voir le résultat, on utilise une instruction de fabrication de chaînes de caractères :

```
make_string 7 `s`;; → string = "sssssss"
```

1.2.3 Recherche d'une décomposition d'un entier en somme d'entiers dont la somme des inverses fasse 1, tel 78 = 2 + 6 + 8 + 10 + 12 + 40

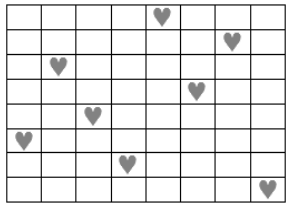
En Lisp, sans se soucier d'un calcul fractionnaire, somme des inverses à 10^{-3} près :

```
(defun decomp (n k s si r) (cond ; s = somme partielle, si = somme des inverses
        ((and (eq s n) (< (abs (1– si)) 0.001)) r)
                ; r est la lite des entiers qui conviennent
        ((or (> s n) (> si 1.001)) nil)
        ((decomp n (1+ k) (+ s k) (+ si (/ 1 k)) (cons k r))) ;  l'entier k est acceptable
        ((decomp n (1+ k) s si r)) )) ; dernier cas, on essaie le successeur de k

Appel par (decomp n 2 0 0 nil)
Exemple (decomp 78 2 0 0 nil) → (40 12 10 8 6 2)

Solution Caml maintenant :
let reel = float_of_int and absr = abs_float;;
let rec interv n = if n = 0 then [] else (n – 1) :: (interv (n – 1));;

let rec cherche s si choisis reste lf eps =
        if reste = [] then if choisis = [] then []
        else (let x = hd choisis
                in cherche (s + x) (si  +. 1./. (reel x)) (tl choisis) (hd lf) (tl lf) eps)
        else (let k = hd reste in if k = s & absr(si–. 1./. (reel k)) <.eps then k :: choisis
        else if k < s then cherche (s – k) (si  –. 1./. (reel k)) (k :: choisis) (tl reste)
                                ((tl reste) :: lf) eps
        else cherche s si choisis  (tl reste) lf eps);;
let main n = cherche n 1. [] (interv n) [] 0.000000001;;
```

Ou encore les mêmes 5 cas, en plus condensé :

```
let rec cherche s si eps = fun [] [] _ -> [] (* échec *)
        | (x::rc) [] (f::lf) -> cherche (s+x) (si +. 1./.(reel x)) eps rc f lf (* remontée *)
        | ch (k::f) lf -> if k = s & absr(si –. 1./.(reel k)) < eps then k::ch (* succès *)
                else if k < s then cherche (s – k) (si –.1./.(reel k)) eps (k::ch) f (f::lf)
                (* cas de la descente dans l'arbre *)
                else cherche s si eps ch f lf;; (* examen du frere *)

let rec sominv = fun [] -> 0. | (x :: r) -> 1./.(reel x) +. sominv r;;
let rec vue l = match l with x :: r -> print_int x; print_string "  ";
                                       vue r | [] -> print_newline();;
for n = 2 to 38 do
        let r = main n in if r <> [] then (print_int n; print_string " -> "; vue r ) done;;

11 -> 2  3  6          24 -> 2  4  6  12       30 -> 2  3  10  15
31 -> 2  4  5  20      32 -> 2  3  9  18       37 -> 2  3  8  24
38 -> 3  4  5  6  20
```

1.2.4 Le problème des reines de Gauss



Ce célèbre problème consiste, encore une fois, à placer n reines sur un damier de n lignes et n colonnes de façon à ce qu'aucune ne soit en position d'être prise par une autre.

On s'arrête à la première solution rencontrée.

Il est recommandé de le faire à la main pour $n = 4$ ou 5.

Pour 3 reines placées aux 3 premières colonnes sur les lignes 6, 3, 5, on représentera cet état provisoire par [5; 3; 6] la position suivante i sera donc empilée en premier.

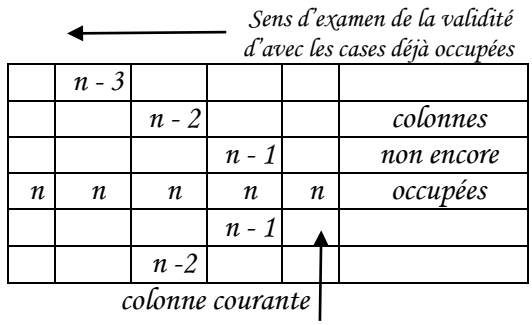
La validité indique que la rangée i est compatible avec l'état antérieur e, à savoir que j colonnes avant celle où on se propose de mettre une reine à la ligne i, si la reine se trouve à la ligne k, on doit avoir $i \neq k$ et sans prise possible en diagonale c'est-à-dire $k \neq i + j$ et $k \neq i - j$. Il faut prendre la liste e des numéros de lignes que peuvent occuper les dames sans être en position de prise mutuellement.

Solution Caml, on définit d'abord la validité :

```
let valide i e = let rec test i j e = match (i, j, e) with (_, _, []) -> true
                        | (i, j, k :: q) -> (i <> k) & (abs(k – i) <> j) & (test i (j + 1) q)
                in test i 1 e;;
```

Pour indiquer que la ligne *i* est valide avec la liste *choix* déjà constituée. Exemple :
valide 5 [3;1];; → true
valide 2 [3;1];; → false

Les états qui suivent e forment la liste fille *f* des états *i :: e* pour les entiers *i* de 1 à *n*, pourvu qu'ils soient valides avec *e*. Les fils suivants un état *e* du damier sont fonction de la dimension *n* de celui-ci.




```
let suiv n e = let rec suiv' i f = if i = n + 1 then f
                        else suiv' (i + 1) (if valide i e then (i :: e) :: f else f)
                in suiv' 1 [] ;;
suiv 8 [5;3;1];; → [[8; 5; 3; 1]; [7; 5; 3; 1]; [2; 5; 3; 1]]
```

Là encore la fonction *but* est anonyme, on indique simplement quand la « solution » arrive à la longueur de la dimension *n* du damier.

```
let reine n = bak (suiv n) (fun x -> n = List.length x) [];;
```

Ainsi par exemple :

```
reine 7;; → (true,  [[]; [7]; [5; 7]; [3; 5; 7]; [6; 3; 5; 7]; [4; 6; 3; 5; 7]; [1; 3; 5; 7];
[6; 1; 3; 5; 7]; [4; 6; 1; 3; 5; 7]; [2; 4; 6; 1; 3; 5; 7]])
```

Mais on peut visualiser un peu mieux la solution :

```
let rec vue e n = if e <> [] then (vue' (hd e); vue (tl e) n)
        where vue' i = (print_string (make_string (i – 1) `-`) ;
        print_string "®"; print_string (make_string (n – i) `-`);  print_newline() ) ;;

let reine n = vue (hd (rev (snd (bak (suiv n) (fun x -> n = list_length x) [])))) n;;
```

Même programme en Lisp
Dans la fonction *fils*, on veut renvoyer les positions possibles suivantes.
La fonction prédéfinie *append* réalise la concaténation entre *nil* qui signifie le faux, mais est en même temps la liste vide, d'où un raccourci permis par ce langage non typé. Le dernier cas est la descente dans l'arbre.

```
(defun reines (n)      (reso n (list n) (list (cdr (fils () n))) (fils (list n) n)))

(defun reso (n ch lf f) (cond
        ((null ch) 'echec) ; cas où l'arbre est entièrement exploré
        ((eq (length ch) n) ch) ; cas final du succès
        ((null f) (reso n (cdr ch) (cdr lf) (car lf))) ; cas d'une remontée
        ((reso n (cons (car f) ch) (cons (cdr f) lf) (fils (cons (car f) ch) n)))))
```

```
(defun fils (ch n) (if (zerop n) nil (append (fbis n 1 ch) (fils ch (1- n)))))
(defun fbis (n p ch) (cond
        ((null ch) (list n))
        ((eq (car ch) n) nil)
        ((eq (+ (car ch) p) n) nil)
        ((eq (– (car ch) p) n) nil)
        (t (fbis n (1+ p) (cdr ch)))))

Exemple (reines 8) → (5 7 2 6 3 1 4 8)
```

SATISFACTION DE CONTRAINTES

Dans les problèmes précédents, la difficulté de modélisation était bien mince, encore que pour les reines de Gauss, on peut représenter l'état du problème par une liste de coordonnées ou par la matrice d'incidence (1 où se trouve une reine, 0 ailleurs) mais ces deux représentations sont visiblement plus mauvaises que celle des numéros de lignes $(l_1, l_2, ...)$ où si $j < i$, chaque l_i vérifiant 3 conditions $l_i \neq l_j$ et $l_i \neq l_j + (i - j)$ et $l_i \neq l_j - (i - j)$.

Dans un problème quelconque, la difficulté réside le plus souvent dans la modélisation et souvent rien ne permet de dire laquelle sera la meilleure avant de se lancer dans la programmation et d'avoir effectué diverses expérimentations. En fait, dans le backtracking, on filtre, c'est-à-dire qu'on réduit l'ensemble des fils d'un nœud en enlevant les fils non valides (consistance d'un nœud), éventuellement ceux invalides, au vu de ce qui a déjà été fait (consistance par arc, par chemin…), idée reprise dans les méthodes « tabou ».

On peut également améliorer le backtrack en réduisant l'espace de recherche suivant un ordre d'examen des fils d'un nœud.

1.2.5 Problème d'Euler du parcours du cavalier sur un échiquier

C'est le problème consistant, sur un échiquier de n cases sur n cases, partant de la case *dx, dy* (comptées à partir de 0), à parcourir tout l'échiquier (une seule fois chaque case) en diagonales de rectangles de 2 sur 3 à chaque fois comme le fait le cavalier sur un jeu d'échec.

Les coups à jouer peuvent être triés suivant le nombre de fils croissant qu'ils ont eux-mêmes (fonction zèbre ci-dessous).

La solution Caml proposée utilise un tableau :

La fonction *cheval* est une fonction qui commence par initialiser un tableau de dimensions n*n à 0 sauf la case dx, dy qui sera affectée par 1 (ce qui se fait en Caml par le symbole <–).

La fonction booléenne *parcours* à la k-ième position (i, j) fixée du cavalier, essaie sa k+1-ième position sur ses voisins potentiels grâce à la fonction booléenne *essai* qui lance *parcours* sur le premier voisin libre en

l'affectant. Laquelle affectation est remise en cause grâce au *or*, au cas où cette suite du *parcours* échoue.

```
let v = [|1; 2; 3; 4|];;   v : int vect = [|1; 2; 3; 4|]
v.(3);; →       int = 4 (* équivalent à vect_item v 3;; *)

let rec filtre p = function [] -> []
(* fonctionnelle donnant la liste des éléments vérifiant une propriété p *)
        | (x :: q) -> if p x then x :: (filtre p q) else filtre p q;;

let voisins i j n = filtre (fun (x, y) -> 0 <= x & x < n & 0 <= y & y < n)
(* liste des au plus 8 coups possibles *)
[(i – 2, j – 1); (i – 2, j + 1); (i – 1, j – 2); (i – 1, j + 2); (i + 1, j – 2); (i + 1, j + 2);
(i + 2, j – 1); (i + 2, j + 1)];;

let affiche tab n = for x = 0 to (n –  1) do for y = 0 to (n – 1) do
        let k = tab.(x).(y)
        in (if k < 10 then print_char ' '); print_int k; print_string " " done;
        print_newline () done; print_newline();;

let cheval n dx dy = let rec ech = Array.make n (Array.make n 0)
        (* k est le dernier numéro placé en i, j *)

and parcours i j k = if k = n*n then (affiche ech n; true)
                                else essai (k+1) (voisins i j n)

and essai e = function [] -> false (* aucun saut possible après *)
                | (x, y)::f -> if ech.(x).(y) <> 0 then essai e f (* case déjà occupée *)
                else (ech.(x).(y) <- e; parcours x y e) or (ech.(x).(y) <- 0; essai e f))
in (ech.(dx).(dy) <- 1; parcours dx dy 1);;
```

Exemples

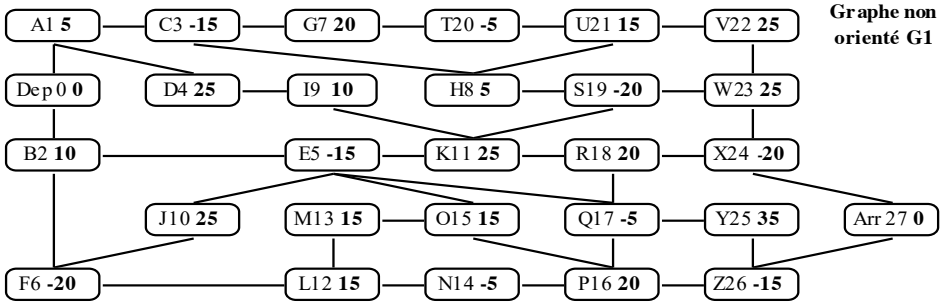
```
cheval 5 0 0;; →
 1  18  5  10  3
 6  11  2 19 14
17 22 13  4  9
12  7  24 15 20
23 16 21  8 25
```

```
cheval 6 0 0 ;; →
  1  8  5  20  3  10
  6 19  2  9  34  21
15 28  7  4  11  32
18 25 16 33 22 35
29 14 27 24 31 12
26 17 30 13 36 23
```

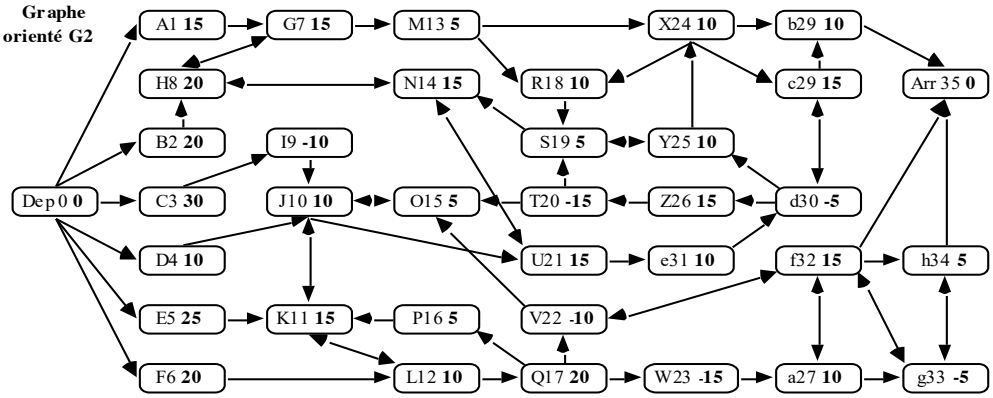
Un problème en vue d'une amélioration de la rapidité d'examen des fils d'un nœud est le suivant : si les fils sont triés suivant le nombre de fils qu'ils ont eux-mêmes, alors la nouvelle fonction modifiée *zèbre* irait-t-elle plus vite ?

1.2.6 Meilleur chemin dans un graphe

Le problème consiste à trouver le chemin entre le départ *Dep* et l'arrivée *Arr* totalisant le plus grand total de points (les points sont en gras). Les sommets peuvent être repérés par une lettre ou un numéro.



Il faut prévoir une représentation du graphe qui puisse servir, ou bien à un graphe non orienté (une relation symétrique) comme *G1* (où on doit trouver 200), aussi bien qu'à un graphe orienté tel que *G2* (où le maximum est 205).



Solution Caml

On représente le graphe comme une liste de listes, chaque sous-liste étant un numéro de sommet (0 pour le départ, 99 pour l'arrivée) et sa valeur suivis des numéros de sommets accessibles.

99 est bien sûr un artifice, on peut trouver autre chose.

Un premier exemple avec le petit graphe g*0* :

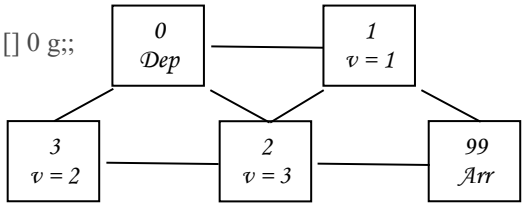
```
let g0 = [[0; 0; 1; 2; 3]; [1; 1; 0; 2; 99]; [2; 3; 0; 1; 3; 99]; [3; 2; 0; 2]; [99; 0; 1; 2]];;
```

```
let g1 = [[0;0;1;2]; [1;5;0;3;4]; [2;10;0;5;6]; [3; –15;1;7;8]; [4;25;1;9];
        [5;–15;2;10;11;15;17]; [6;–20;2;10;12]; [7; 20;3;20]; [8;5;3;19;21];
        [9;10;4;11]; [10;25;5;6]; [11;25;5;9;18;19]; [12;15;6;13;14]; [13;5;12;15];
        [14;–5;12;16]; [15;15;5;13;16]; [16;20;14;15;17;26]; [17;–5;5;16;18;25];
        [18;20;11;17;24]; [19;–20;8;11;23]; [20;–5;7;21]; [21;15; 8;20;22];
        [22; 25; 21;23]; [23;25;19;22;24]; [24;–20;18;23;99]; [25;35;17;26];
        [26;–15;16;25;99]; [99;0]];;

let g2 = [[0; 0; 1;2;3;4;5;6]; [1; 15; 7]; [2; 20; 8]; [3; 30; 9]; [4; 10; 10]; [5; 25; 11];
        [6; 20; 12]; [7; 15; 8;13]; [8; 20; 7;14]; [9; –10; 10;15]; [10; 10; 11;21];
        [11; 15; 10;12]; [12; 10; 11;17]; [13; 5; 18;24]; [14; 15; 8;21]; [15; 5; 9];
        [16; 5; 11]; [17; 20; 16;22;23]; [18; 10; 19]; [19; 5; 14;25]; [20; –15; 15;19];
        [21; 15; 31]; [22; –10; 15;32]; [23; –15; 27]; [24; 10; 18;28;29];
        [25; 10; 19;24]; [26; 15; 20]; [27; 10; 32;33]; [28; 10; 99]; [29; 15; 28;30];
        [30; –5; 25;26;29]; [31; 10; 30]; [32; 15; 22;27;33;34;99]; [33; –5; 27;32];
        [34; 5; 99]; [99; 0]];;

let rec fils s ll = if s = hd (hd ll) then tl (tl (hd ll)) else fils s (tl ll);;
        (* renvoie la liste des sommets accessibles depuis s *)
let rec vue = fun [] -> print_string " somme = "; ()
        | (x::q) -> (print_int x; print_string " "; vue q);;
let rec val s = fun [] -> 0
        | (x::q) -> if s = hd x then hd (tl x) else val s q;; (* valeur de s dans g *)
```

On définit une fonction récursive terminale sans boucle, ni variables locales :

```
let rec explo ch fr sp fi sol sm g =
(* ch chemin, fr liste des listes des frères restant à explorer pour chq noeud de ch *)
        if ch = [] then (vue (rev sol); sm)
        else if hd ch = 99 then explo (tl ch) (tl fr) (sp – val (hd ch) g) (hd fr)
                                        (if sm < sp then ch else sol) (max sp sm) g
        else if fi = [] then explo (tl ch) (tl fr) (sp – val (hd ch) g) (hd fr) sol sm g
        (* retour *)
        else if mem (hd fi) ch then (explo ch fr sp (tl fi) sol sm g)
        (* boucles interdites *)
        else (explo ((hd fi) :: ch) ((tl fi) :: fr) (sp + (val (hd fi) g)) (fils (hd fi) g)
                        sol sm g);; (* examen du noeud suivant *)
```

```
let jeu g = explo [0] [[]] 0 (fils 0 g) [] 0 g;;
```



Pour les graphes considérés :
jeu g0;; → 0 3 2 1 99 somme = 6

jeu g1;;
→ 0 2 5 10 6 12 13 15 16 26 25 17 18 11 9 4 1 3 7 20 21 22 23 24 99 somme = 200
jeu g2;;
→ 0 6 12 17 16 11 10 21 31 30 25 19 14 8 7 13 24 29 28 99 somme = 205

1.3 L'ALGORITHME A*

Avant d'exposer d'autres techniques de résolution de problèmes se traduisant par des arbres, rappelons qu'on nomme habituellement algorithmes les méthodes de calcul dont on a prouvé qu'elles fournissaient le résultat escompté, en opposition à « heuristique » pour des méthodes non prouvées. En fait, le terme désigne dans un sens plus restreint un moyen quelconque pour guider les choix et réduire les alternatives et plus précisément encore souvent une fonction de coût que l'on va chercher à minimiser.

L'algorithme A* consiste, pour un parcours d'arbre, à considérer à chaque instant une liste de chemins G.

Au départ G n'est que le chemin de longueur 0 constitué par la racine (l'état initial).
Jusqu'à ce que G soit vide ou que le but soit atteint, on regarde le premier chemin de la liste.

S'il arrive au but c'est fini (succès).

Sinon, il est retiré et remplacé par les chemins allant jusqu'à ses fils.

G est trié par coûts croissants (branch and bound) ou bien par le coût du chemin ajouté à l'estimation du coût h pour rejoindre le but.

De plus, si plusieurs chemins aboutissent au même nœud, on élimine de G tous ceux de coût supérieur.

Notations : $k^*(u, v)$ est le coût minimal d'un chemin entre les états u et v
$g^*(u)$ = *coût minimal de l'état initial* à l'état u, $h^*(u)$ = *coût minimal de* u à un état terminal
$f^*(u) = g^*(u) + h^*(u)$ est donc le meilleur coût d'une solution passant par u.

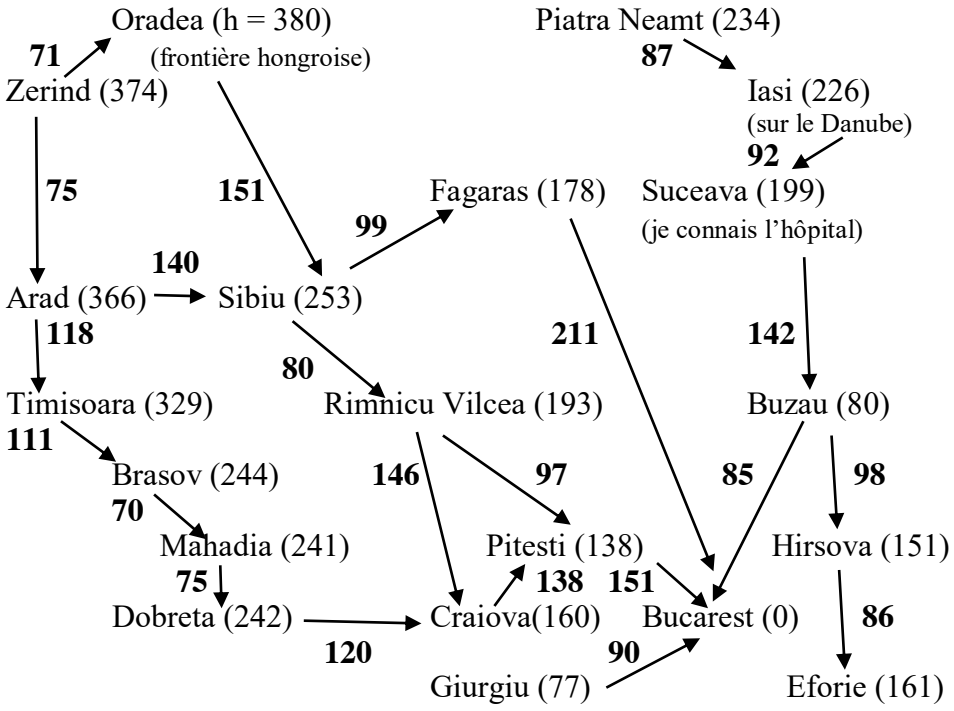
Comme en chaque état u, ces fonctions ne sont pas connues, on tâche de les estimer, on estime notamment h^* par une « heuristique » h telle que l'on ait $h = 0$ sur tous les états terminaux (h doit être « coïncidente »). On dit que :
h minorante si et seulement si $h \leq h^*$ (sinon, il se nomme algorithme A). Si n est le nombre d'états, dans le pire des cas, la complexité est en 2^n, mais si h est minorante, A* a une complexité en n^2.

On montre que h est monotone si et seulement si pour tous u et v, on a l'inégalité $h(u) - h(v) \leq k(u, v)$, alors A* a une complexité en n.

1.3.1 Exemple roumain tiré d'un ouvrage célèbre (Russell – Norvig)

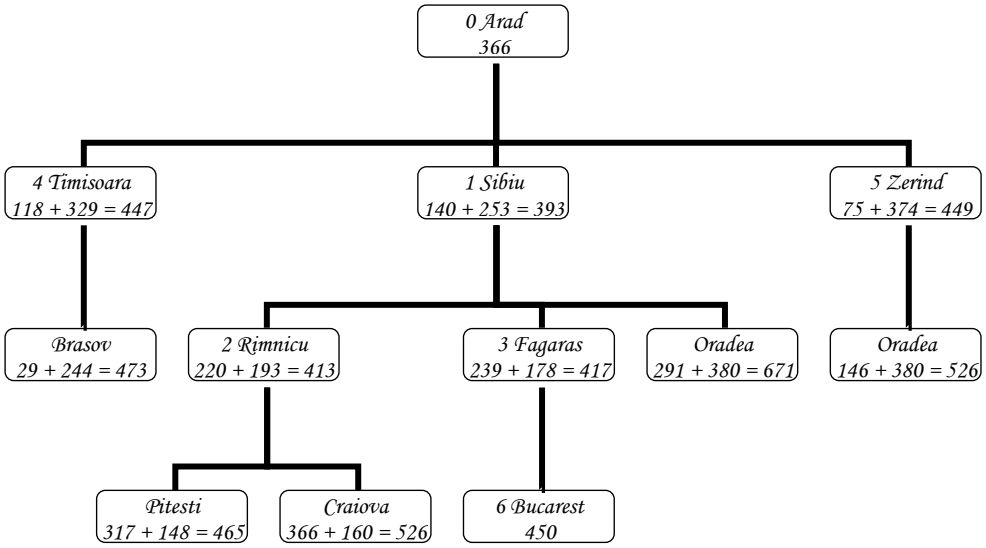
On donne sur le graphe les distances entre villes et associée à chaque ville une « heuristique » constituée par une estimation de la distance à Bucarest. On suppose donc non connue cette distance mais seulement quelques distances localement.

On voit pour un graphe plus important, comme celui d'un jeu, l'analogie avec l'idée que l'on peut se faire des coups plus ou moins gagnants. Partant d'Arad, le but est de développer à la main l'algorithme A* pour rejoindre Bucarest.



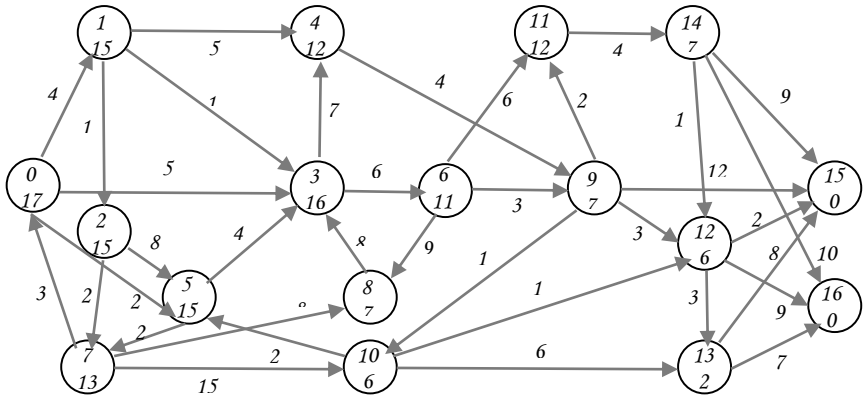
Le début du développement de l'arbre se fait en examinant, après les nœuds-fils d'un nœud, tout le re-calcul de l'heuristique en fonction de la distance réelle parcourue, pour ensuite reclasser à nouveau toutes les branches existantes et repartir avec l'examen des fils de la plus courte branche.

Dans cet exemple, on a ici seulement 6 branches examinées avant d'arriver au but et on remarquera que l'on peut arriver à la même ville par plusieurs chemins.



1.3.2 Exemple de Farreny-Ghallab

Chaque nœud du graphe est numéroté de 0 à 16 et entaché d'une heuristique estimant le coût pour arriver en fin du parcours. Le départ est en 0, l'arrivée en 15 ou 16, les arcs sont valués.



La solution est assez longue, car en 12 étapes, et on doit trouver le chemin (0, 1, 4, 9, 10, 12, 13, 16) de coût 25.

L'ALGORITHME AO*

Dans le cas d'un problème se traduisant en sous-problèmes par un graphe *et* / *ou*, c'est un algorithme de style A* de complexité coûteuse où chaque fois qu'un successeur d'un état est envisagé, celui-ci perturbe le coût de l'antécédent et donc des ancêtres. A chaque fois les coûts sont révisés et le tri se fait donc sur de nouveaux coûts *[Martelli 1973]*.

Soit G un graphe *et* / *ou* sans circuits, contenant un état initial u_0, des états terminaux T et soit h une heuristique.

On souhaite trouver une solution partielle G' contenant u_0 et telle que pour tout état v de G', c'est une feuille ou bien il y ait un connecteur unique amenant à des fils dans G'. Si les feuilles de G' sont des états de T alors on a une solution complète.

On note par ailleurs $I(u)$ l'ensemble des opérateurs qui peuvent s'appliquer en u et $k(u, i)$ le coût de l'opérateur i.

Soit P l'ensemble des états engendrés mais dont on n'a pas encore développé les fils, P est $\{u_0\}$ initialement.

Q est l'ensemble de tous les états développés (initialement vide)

$f(u)$ est le coût minimal provisoire de u (initialement $G' = \{u_0\}$ et $f(u_0) = h(u)$)

Tant qu'il y a dans G' des états pendants (non développés ou non terminaux) :

Soit u un sommet pendant de G' faire $P \leftarrow P-\{u\}$; $Q \leftarrow Q + \{u\}$

Si on ne peut pas décomposer u alors $f(u)$ est infini

sinon Pour tout $i \in I(u)$ et tout état v obtenu par $i(u)$ (i est un connecteur *et*/*ou*)

$P \leftarrow P+\{v\}$; $f(v) = h(v)$

Calculer $f_i(u) = k(u, i) + \sum\{f(v) / v \in i(u)\}$

$f(u)$ est le min des $f_i(u)$ pour $i \in I(u)$

Soit A l'ensemble des pères de u

Pour tout v dans A sans successeur dans A

$A \leftarrow A - \{v\}$ on recalcule $f(v) = min\ \{k(v, i) + \sum\{f(w) / w \in i(v)\} / i \in I(v)\}$

$A \leftarrow A + pères(v)$ (et on modifie donc f pour les ascendants de v)

1.3.3 Exemple : décrire la décomposition de 6

Avec les règles 6 → 3 + 3; 4 → 2 + 2; 3 → 2 + 1; 2 → 1 + 1; 6 → 4 + 2; 4 → 3 + 1. On souhaite n'avoir que des 1, le coût de 1 étant 0, sinon, le coût du nœud n est n. Dès qu'on examine un fils d'un nœud donnant lieu à k nœuds, le coût du père est révisé en k + ∑cout(fils). Le coût du connecteur *« et »* est fixé à 2. Développer toutes les étapes en traçant à chaque fois l'arbre partiel arrivant à la valeur 6.

A la première étape 6 possède 2 fils de coûts 8 et 8 donc équivalents.
On développe donc le premier (6 = 3 + 3), son coût révisé devient 4, celui de 6 devient 9.
Etape 3, comme 9 > 3, on développe l'autre (6 = 4 + 2) en débutant par 4, son coût devient min(6, 5) = 5, celui de 6 passe à 9.
Etape 4, on garde cette solution car non supérieure à l'autre :
6 (coût 10) = (2 + 4) ; 4 (coût 6) = 2 + 2 (coût 6) ou 3 + 1 (coût 6) ;
et 3 = 2 + 1 (coût 4)
Etape 5, comme 10 > 9, on revient à la branche 3 + 3, le coût de 6 est maintenu à 9 et on développe le second 3, ce qui révise le coût à 10
On reste sur cette solution de coût 10, soit 6 = 3 +3 = (1 + 2) + (1 + 2)
= (1 + (1 +1)) + (1 + (1 + 1))

1.4 L'ALGORITHME MINIMAX

Cet algorithme est du à Von Neumann (1903-1957) qui fut avec A. Turing, un des fondateurs de l'informatique théorique. Il s'applique pour tout jeu où deux joueurs choisissent à tour de rôle un coup parmi plusieurs coups possibles.

Si les chances d'un « état » du jeu sont évalués par une certaine fonction heuristique mesurée positivement lorsqu'il favorise un des joueurs (nommé *max*) et négativement pour son adversaire (nommé *min*) alors l'algorithme consiste à faire un choix sur la plus grande valeur d'une fonction *minimax* lorsque c'est à *max* de jouer, (car c'est ce qu'il jouerait pour gagner c'est-à-dire perdre le moins) et sur la plus petite (algébriquement, car c'est ce que *min* jouerait pour perdre le moins) lorsque c'est à *min* de jouer.

Cette fonction est définie récursivement par ce qui vient d'être dit, et par l'heuristique lorsqu'elle arrive sur un état terminal ou au niveau de profondeur fixé à l'avance.

La fonction *minimax* peut être définie, à condition que Max soit la fonction *max* et Min la fonction *min*, par :

minimax (arbre, joueur) =
si *arbre* réduit à une feuille alors *arbre*
sinon appliquer la fonction *joueur*
sur la liste des résultats de *minimax (adversaire de joueur)*
sur tous les fils de *arbre*

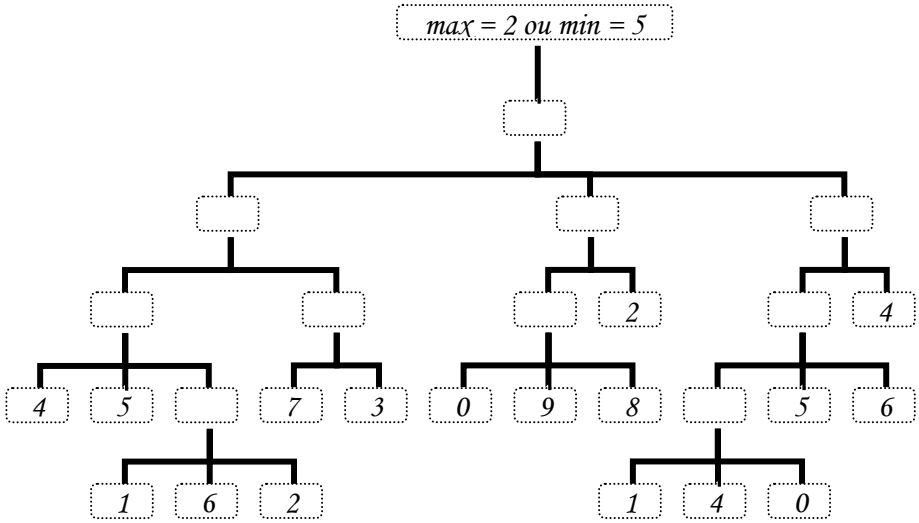
1.4.1 Minimax en profondeur non limitée programmé en Lisp

Réaliser l'évaluation *minimax* d'un arbre dont les nœuds sont exactement des valeurs numériques (si le nœud est une feuille), on renvoie sa valeur, s'il a une profondeur paire, on renvoie le *max* des valeurs de ses fils, et s'il est à une profondeur impaire, on renvoie le *min* de ses fils (on convient que la racine est au niveau 0).

Dessiner en premier lieu l'arbre représenté par la liste qui suit et calculer la valeur (pour *max*, comme pour *min*) de :

((((4 5 (1 6 2)) (7 3)) ((0 9 8) 2) (((1 4 0) 5 6) 4)))

Programmer la valeur d'un tel arbre.



Une très belle solution avec deux fonctions mutuellement récursive d'un seul argument, l'arbre A :

```
(defun valmax (A) (if (atom A) A (apply 'max (mapcar 'valmin A))))
        ; à utiliser à la racine
(defun valmin (A) (if (atom A) A (apply 'min (mapcar 'valmax A))))
```

Ou encore une solution astucieuse avec une seule fonction de deux arguments à condition que le joueur j soit appelé *max* ou *min* :

```
(defun minimax (a j) (if (atom a) a
        (apply j (mapcar (lambda (x) (minimax x (if (eq j 'max) 'min 'max))) a)))
```

Plus précisément, en général :

```
fonction minimax (e : etat, j : joueur, n : niveau de profondeur souhaité)
                retourne : une valeur permettant de classer les coups à jouer
si e est un succes pour max, retourne +∞
si e est un succes pour min, retourne –∞
si e est un match nul (le jeu est bloqué), retourne 0
si n = 0, minimax = heuristique(e)
si joueur = max alors
        minimax = max{minimax(e', min, n–1) / e' coup pour max à partir de e}
sinon (joueur = min)
        minimax = min{minimax(e', max, n–1) / e' coup pour min à partir de e}
```

LES COUPES ALPHA-BETA DANS LE CALCUL DU MINIMAX

Aux étapes *min*, si un nœud est connu, dès que l'on rencontre un neveu inférieur, il est inutile d'explorer les autres neveux car leur père aura donc un *min* plus petit que celui de son frère déjà connu et donc il ne sera pas choisi par *max* à l'étape supérieure. On peut donc élaguer l'arbre, c'est-à-dire