```
======================================================================
I. Presentacion del curso
======================================================================

Evaluacion:
  3 mini proyectos y proyecto final de 50%,
  Examen de 50%.
  Grupos de 3 personas.

Horas de consulta:
  Acercarse a la oficina, con o sin cita.

Alcance y objetivos del curso. Temas:
  Busqueda:
    Busqueda heuristica en grafos OR:
      BFS, DFS, DFID, A*, IDA*, LRTA*, Branch and bound

    Descomposicion de problemas:
      Grafos AND/OR
      AO*
      Ciclos
      LDFS

    Arboles de Juego
      Minmax
      Alpha-beta prunning
      Scout, MTD
      UCT

  Planificacion Automatica
    STRIPS y SAS+
    (Ver libro)

  CSPs
    Varios algoritmos y tecnicas
    Nociones de consistencia
    Treewidth, strong consistency, backtrack-free search

  Rep. conocimiento e inferencia en logica:
    Logica Proposicional:
      CNF, Complejidad, 2CNF
      DPLL, Clause Learning
      Caso de Estudio: Planificacion via SAT

    Logica de Primer Orden
      Forma Clausal, Resolucion

  Manejo de Incertidumbre
    Representacion e inferencia:
      Markov Networks
      Redes Bayesianas
      Factor Graphs

----------------------------------------------------------------------

?Que es la I.A.?
  Un poco de historia
  Overview of subfields including
    Vision vs computacion grafica
    NLP vs speech generation
    Busqueda & Planning
    Machine learning
    etc

Lenguajes de programacion
  Lisp, Scheme precursores de programacion funcional
  Prolog
  Dedicados como sistemas expertos (rules), CP y otros
  Generales por motivos de eficiencia: C/C++, Java, Python, etc.
```



```
----------------------------------------------------------------------

Busqueda:

  Espacio de busqueda caracterizado por:
    - conjunto finito de estados
    - estado inicial s_0
    - estados objetivos G
    - operadores de transicion (acciones) A, y A(s)
    - funcion de transicion f(s,a)
    - funcion de costo c(s,a)

  Representacion:
    explicita (mat. incidencia/lista adyacencia) vs implicita

  Ejemplo: agent que navega en grid n x m

  Soluciones: que es solucion, que es solucion optima.

  Otros ejemplos:
    8-puzzle, 15-puzzle, 24-puzzle, (n^2 - 1)-puzzle, (nm - 1)-puzzle
    Cubo de Rubik
    Torres de Hanoi con 4 astas

  Nociones fundamentales:
    - estados vs nodos que representan estados
    - duplicados
    - arbol de busqueda: ciclos de estados generan arboles infinitos
    - generar: estado se genera cuando se crea un nodo que lo representa
    - expandir: estado se expande cuando se generan todos sus sucesores

  Algoritmos: BFS, DFS, DFID, A*, IDA*, LRTA*, BnB

  Dimensiones de analisis:
    completitud, optimalidad, complejidad tiempo y espacio

======================================================================
II. Busqueda: preliminares, BFS, DFS y DFID.
======================================================================

?Que contiene un node en el arbol de busqueda?

Cada nodo en el arbol de busqueda representa un estado
en el espacio de busqueda mas informacion acerca de donde,
cuando y como los estados son encontrados durante la busqueda.

Todo nodo n tiene asociado:

state(n) = estado representado por n
parent(n) = pointer al nodo desde el cual n es alcanzado
action(n) = accion que mapea state(parent(n)) en state(n)
g(n) = cost del camino representado por n

Los algoritmos de busqueda utilizan las siguientes operaciones
sobre estados:

- init() que genera el estado inicial
- is-goal(s) que prueba si s es un estado objetivo
- succ(s) genera una lista de estados sucesores de s junto
  con las acciones que los alcanzan. La lista es una lista
  de pares <o,s> donde o es una accion (op.) y s un estado.

y las siguientes operaciones sobre nodos:

- make-root-node(s) construye un nodo raiz del arbol de busqueda
  que representa al estado dado:

  make-root-node(s):
```

```
    n := new node
    state(n) := s
    parent(n) := null
    action(n) := null
    g(n) := 0
    return n

- make-node(n,a,s): construye un nodo que representa al estado
  s dado generado por la accion a aplicada en state(n)

  make-node(n,a,s):
    n' := new node
    state(n') := s
    parent(n') := n
    action(n') := a
    g(n') := g(n) + cost(state(n),a)

- extract-solution(n): extrae el camino que conlleva al nodo n

  extract-solution(n):
    solution := new list
    while parent(n) != null do
        solution.push-front(action(n))
        n := parent(n)
    return solution

------------------------------------------------------------------------

Breadth-First Search (busqueda en amplitud)

  Esquema general sin/con eliminacion de duplicados

  BFS (sin eliminacion de duplicados)
  -----------------------------------
  queue := new fifo-queue
  queue.push-back(make-root-node(init()))
  while !queue.empty() do
      n := queue.pop-front()
      if is-goal(state(n)) then
          return extract-solution(n)
      for each <o,s> in succ(state(n)) do
          n' := make-node(n,o,s)
          queue.push-back(n')
      return UNVOLSABLE

  Mejoras posibles:
    Eliminacion de duplicados
    Chequear por goal al momento de generacion

  BFS (con eliminacion de duplicados)
  -----------------------------------
  queue := new fifo-queue
  queue.push-back(make-root-node(init()))
  closed := emptyset
  while !queue.empty() do
      n = queue.pop-front()
      if state(n) isn't in closed, then
          closed.insert(state(n))
          if is-goal(state(n)) then
              return extract-solution(n)
          for each <o,s> in succ(state(n)) do
              n' := make-node(n,o,s)
              queue.push-back(n')
  return UNVOLSABLE

  Analisis
  --------
  Completitud: Suponga que existe un estado objetivo en el espacio de
  busqueda. Sea \pi un camino desde el estado inicial a un estado goal.
```



```
  Invariante-1: siempre existe un nodo n en queue tal que state(n)
  aparece en \pi y state(n) isn't closed (para BFS con elim. de dup.)

  Dem: por induccion en el numero de iteraciones del lazo. Al comienzo
  el invariante es cierto ya que queue solo contiene el estado inicial.
  Suponga que el invariante es cierto al inicio del ciclo. Si el invariante
  se cumple para un nodo n' en queue distinto del nodo seleccionado n,
  entonces el invariante se sigue cumpliendo al inicio de la proxima
  iteracion. Consideramos los dos algoritmos de BFS.

  BFS sin eliminacion de duplicados. Si state(n) es goal, entonces
  el algoritmo termina. Sino, todos los sucesores de state(n) son
  generados e insertados en queue. Por lo tanto, el sucesor de
  state(n) en \pi es insertado en queue, y el invariante se cumple
  al inicio de la proxima iteracion.

  BFS con eliminacion de duplicados. Por definicion del invariante,
  state(n) no esta closed ya que por hipotesis n es el unico nodo
  que satisface el invariante. Si state(n) es goal, el algoritmo
  termina. Sino, todos sus hijos son generados e insertados en queue.
  Tenemos que mostrar que despues de las inserciones, el invariante
  se cumple. ... TODO

  Consecuencia: BFS (con/sin eliminacion de duplicados) es completo.

  Optimalidad: BFS consigue un camino de menor numero de pasos (no
  necesariamente de menor costo), si dicho camino existe.

  Invariante 2: los estados en queue estan ordenados por la distancia
  al node raiz. Mas aun, la cola solo contiene estados a distancia
  n o n+1 de la raiz, para algun n.

  Dem: TODO

  Demostracion de optimalidad por reduccion al absurdo. Suponga que
  BFS consigue un camino que no es de menor longitud. Al momento de
  terminar, se ha extraido un nodo n de queue (un nodo goal) tal que
  su costo es mayor al costo del camino optimo. Por Inv-1, en queue
  hay un nodo n' en el camino optimo. Por lo tanto, g(n') < g(n).
  Por Inv-2, n' se debio extraer antes que n.

  Para analisis de tiempo y espacio. Asuma un arbol de busqueda
  regular con factor de ramificacion b y donde el primer objetivo
  encontrado esta a profundidad d.

  Analisis de Tiempo: O(b^d)

  Analisis de Espacio: O(b^d)

------------------------------------------------------------------------

Depth-First Search (busqueda en profundidad)

  DFS (recursivo)
  ---------------
  n := make-root-node(init())
  <plan,cost> := DFS-Search(n, 0)
  return plan

  DFS-Search(node n, int g)
  -------------------------
  if is-goal(state(n)) then return <extract-solution(n), g>
  for each <a,s> in succ(state(n)) do
      <plan,cost> := DFS-Search(make-node(n,o,s), g + cost(state(n),a)
      if plan != null then return <plan,cost>
  return <null, infty>

  Analisis: incompleto, tiempo = O(b^d), espacio = O(bd)
```

```
Depth-First Iterative Deepening (DFID)

  DFID (recursivo)
  ----------------
  n := make-root(init())
  t := 0
  plan := null
  while plan == null do
      <plan,cost> := Depth-Bounded-DFS(n, 0, t)
      if plan != null then return plan
      t := cost
  return null

  Depth-Bounded-DFS(node n, int g, int t)
  ---------------------------------------
  if g > t then return <null,g>
  if is-goal(state(n)) then return <extract-solution(n), g>

  new-t := infty
  for each <a,s> in succ(state(n)) do
      <plan,cost> := Depth-Bounded-DFS(make-node(n,a,s), t, g + cost(state(n),a)
)
      if plan != null then return <plan,cost>
      new-t := min(new-t, cost)
  return <null, new-t>

  Analisis: completo, optimo para funcion de costo uniforme,
  tiempo = O(b^d), espacio = O(bd)

======================================================================
III. Busqueda: UCS, Best-First Search y A*
======================================================================

Uniform-cost Search: best-first search c/ eliminacion duplicados

  UCS (con eliminacion de duplicados)
  -----------------------------------
  queue := new priority-queue (sorted by g-value)
  queue.insert(make-root-node(init()))
  closed := emptyset
  while !queue.empty() do
      n = queue.pop-first()
      if state(n) isn't in closed, then
          closed.insert(state(n))
          if is-goal(state(n)) then
              return extract-solution(n)
          for each <o,s> in succ(state(n)) do
              n' = make-node(n,o,s)
              queue.push-back(n')
  return UNVOLSABLE

Analisis: complete, optimal, tiempo y espacio O(b^n)

----------------------------------------------------------------------

Busqueda con heuristicas:

Heuristica h(.) es funcion sobre estados que "estima" el costo del
camino de menor costo desde un estado dado a un goal. Las heuristicas
se aplican sobre nodos n, como h(n)=h(state(n)), y si queremos ser
mas generales, se pueden definir directamente sobre los nodos

Estimacion de la distancias:

1. For most heuristic search algorithms, h does not need to
   have any strong properties for the algorithm to be correct
   or complete,
```



```
2. However, the efficiency of the algorithm closely relates
   to how accurately h reflects the actual goal distance.

3. For some algorithms, like A*, we can prove strong formal
   relationships between properties of h and properties of the
   algorithm (optimality, dominance, run-time for bounded error,
   etc)

4. For other search algorithms, "it works well in practice" is
   often as good an analysis as one gets.

Propiedades:

A heuristic h is called:

1. safe (segura) if h*(n)=infty for all n with h(n)=infty
2. goal-aware (reconoce el goal) if h(n)=0 for all goal nodes n
3. admissible if h(n) <= h*(n) for all nodes n
4. consistent if h(n) <= h(n') + c(state(n),a) for all
   nodes n, n' such that <a,state(n')> in succ(state(n))

Implicaciones: (4) => (3) => (1)

Heuristica Perfecta h*:

The optimal or perfect heuristic of a search space is the
heuristic h* which maps each search node n to the length of a
shortest path from state(n) to any goal state. Note: h*(n)
equals infinity iff no goal state is reachable from n.


----------------------------------------------------------------------


Idea: utilizar la heuristica h(.) p/ seleccionar proximo nodo
      a expandir. Resultado: Greedy Best-First Search


  GBFS (con eliminacion de duplicados)
  ------------------------------------
  queue := new priority-queue sorted by h-value
  queue.insert(make-root-node(init()))
  closed := emptyset
  while !queue.empty() do
      n = queue.pop-first()
      if state(n) isn't in closed, then
          closed.insert(state(n))
          if is-goal(state(n)) then
              return extract-solution(n)
          for each <o,s> in succ(state(n)) do
              n' = make-node(n,o,s)
              if h(n') < infty:
                  queue.push-back(n')
  return UNVOLSABLE

Analisis: completo (para h segura por deteccion de duplicados), suboptimo,
          tiempo y espacio igual a BFS (para h=0)

----------------------------------------------------------------------


Como incorporar la heuristica en un algoritmo que sea completo
y optimo? Respuesta: Best-First Search.

  BFS (con eliminacion de duplicados)
  -----------------------------------
  queue := new priority-queue sorted by g+h-value
  queue.insert(make-root-node(init()))
  closed := emptyset
  while !queue.empty() do
      n = queue.pop-first()
```

```
      if state(n) isn't in closed or g(n) < distance(state(n)), then
          closed.insert(state(n))
          distance(state(n)) = g(n)
          if is-goal(state(n)) then
              return extract-solution(n)
          for each <o,s> in succ(state(n)) do
              n' = make-node(n,o,s)
              if h(n') < infty:
                  queue.push-back(n')
  return UNVOLSABLE

Nota: Cuando el node n se expande aun cuando state(n) esta cerrado (porque
      g(n) < distance(state(n))), se dice que el estado state(n) se re-abrio.

El valor f de un nodo n es f(n) = g(n) + h(n)

Analisis:

1. Completo para h segura (aun sin eliminacion de duplicados)
2. Optimo si h es admissible
3. Nunca re-abre un nodo si h es consistente
4. Tiempo / espacio en peor caso igual a UCS (caso h=0)

Implementacion:

1. In the heap-ordering procedure, it is considered a good
   idea to break ties in favour of lower h values

2. Can simplify algorithm if we know that we only have to
   deal with consistent heuristics

3. Common, hard to spot bug: test membership in closed at
   the wrong time


Ejemplo: 15-puzzle, Manhattan distance: def, prop., computo

========================================================================
IV. Busqueda: Weighted A*, IDA*, HC y EHC
========================================================================

Weighted A*: igual a A* pero con la heuristica multiplicada por
             constante W

Rol de W:

1. Para W=0, se convierte en UCS
2. Para W=1, se convierte en A*
3. Para W -> infty, se convierte en Greedy Best-First Search

Propiedad:

For W > 1, can prove similar properties to A* , replacing
optimal with bounded suboptimal: generated solutions are
at most a factor W as costly as the optimal ones.

------------------------------------------------------------------------

Linear-space A* = IDA*

DFID es para BFS lo que IDA* es para A*.


  IDA* w/ heuristic h
  -------------------
  n := make-root-node(init())
  t := f(n) [= h(n)]
  plan := null
  while plan == null do
```



```
      <plan, cost> := Bounded-DFS(n,t,0)
      if cost == infty then return null
      t := cost
  return plan


  Bounded-DFS(node n, int t, int g)
  ---------------------------------
  f := g + h(n)
  if f > t then return <null, f>
  if is-goal(state(n)) then return <extract-solution(n), g>

  new-t := infty
  for each <a,s> in succ(state(n)) do
      <plan, cost> := Bounded-DFS(make-node(n,a,s), t, g + cost(s,a))
      if plan != null return <plan, cost>
      new-t := min(new-t, cost)
  return <null, new-t>


------------------------------------------------------------------------

Other algorithms for finding quick solutions: Hill-Climbing and
Enforced Hill-Climbing

  Hill-Climbing
  -------------
  n := make-root-node(init())
  while true do
      if is-goal(state(n)) then return extract-solution(n)
      Succ := { make-node(n,a,s) : <a,s> in succ(state(n)) }
      n := extract node from Succ minimizing h (random-tie breaking)

  Properties:

  1. Different from GBS as GBS select the best in queue with evaluation
     function f(n) = h(n), while HC selects the best child.

  2. Can get stuck in cycle or local minima

  3. Many variations including tie-breaking strategies, restarts, etc

  Variation: Enforced Hill-Climbing: procedure improve make a breadth-first
  search to find a more promising node than n0

  Improve(n0)
  -----------
  queue := new fifo-queue
  queue.push(n0)
  closed := emptyset
  while !queue.empty() do
      n := queue.pop-first()
      if state(n) isn't in closed then
          closed.insert(state(n))
          if h(n) < h(n0) then return n
          for each <a,s> in succ(state(n)) do
              n' := make-node(n,a,s)
              queue.push-back(n')
  return null

  Enforced Hill-Climbing
  ----------------------
  n := make-root-node(init())
  while n != null and !is-goal(state(n)) then
      n := Improve(n)
  return n == null ? null : extract-solution(n)

  Properties:
```

```
  1. a very popular algorithm when one just wants a solution and
     have a reasonable heuristic

  2. improve can fail due to either there is no solution from n0
     (dead end) or run out of memory

  3. complete for undirected search spaces (where the successor
     relation in symmetric) if h(n) = 0 for all and only goal nodes


======================================================================
V. Busqueda: Branch-and-Bound, LRTA*. Heuristicias
======================================================================

  Branch-and-bound
  ----------------
  alpha := infty
  best-leaf := null
  DFS-BnB(make-root-node(init()))
  return <extract-solution(best-leaf),alpha>

  DFS-BnB(n)
  ----------
  if f(n) > alpha then return null
  if is-leaf(n) then
      if g(n) < alpha then
          best-leaf := n
          alpha := g(n)
      end if
  end if

  for each <a,s> in succ(state(n)) do
      DFS-BnB(make-node(n,a,s))

  Works for problems in which all solutions appear as leafs of the tree
  and there are no infinite branches

  Analisis: completo, optimo, linear space, exponential time

  Heavily used in some combinatoria problems such as TSP,
  some cases can deliver best performance than IDA*

----------------------------------------------------------------------

  Real-Time Search
  ----------------

  Setting: agent acting (moving) in a real or simulated environment
  that wants to reach a goal state. Interleaves "planning" and execution.
  Assume that the environment is safely explorable.

  Maintains a global hash table that is used to store values associated
  with nodes that are updates each time the agent makes a decision.

  LRTA* stands for "Learning Real-Time Search"

  LRTA*
  -----
  n := make-root-node(init())
  repeat
    LRTA*-Trial(n)
  until [some condition]

  LRTA*-Trial(n)
  --------------
  while !is-goal(n) do
      for each <a,s> in succ(state(n)) do
          next[a] := s
          Q(state(n),a) := c(state(n),a) + H(s)
```



```
      Select a* as action a minimizing Q(state(n),a)
      Update H(state(n)) := Q(state(n),a*)
      Set n := make-node(n,a*,next[a*])

  H(.) is a hash table that is initially empty. When fetching a value
  for s, if H(s) does not contain an entry for s, the value h(s) is
  returned, else H(s) is returned. when updating H(s), if H contains
  a entry for s, the value of the entry is changed. Otherwise, an
  entry for s is allocated and initialized to the value.

  Properties: on safely explorable environments, each LRTA* trial
  terminates in finite time, and the values in the hash table
  eventually converge. If, in addition, h is admissible, the  value
  for the "relevant states" converge to their optimal values, and
  the trials eventually "behave" optimal.

----------------------------------------------------------------------

  ?De donde vienen las heuristicas?

  Relajaciones del problema que pueden resolverse rapidamente:
  polinomialmente (en teoria), linealmente o cuadraticamente (practica)

  Ejemplo: 15-puzzle
  ------------------

  Admisible porque Manhattan distance is lower bound on *total*
  number of movements that each tile must move. Also, cast it as
  a relaxation where the tiles can move over each other.

  Abstraction heuristics
  ----------------------

  Create abstraction either by collapsing nodes, reducing costs,
  or adding edges (shortcuts). Abstractions can be either implicit
  or explicit, and solved off-line or on-line.

  A special type of abstraction, pattern database, is an explicit
  and off-line heuristic that had been quite successful.

  Pattern Databases
  -----------------

  Example: 15-puzzle: definition, how to compute it, etc.

======================================================================
VI. Descomposicion de problemas. AO*.
======================================================================

Descomposicion de problemas:

Frecuentemente, un problema se descompone en subproblemas
que tienen que resolverse. Tipico de problemas que pueden
ser resueltos por estrategias divide-and-conquer. Las
dependencias entre problemas y subproblemas pueden
representarse como un arbol AND/OR.

Ejemplo: Torres de Hanoi con 3 postes

Move(1..n,1,3) decomposes into
    Move(1..n-1,1,2) & Move(n,1,3) & Move(1..n-1,2,3)

Move(1..n-1,1,2) decomposes into
    Move(1..n-2,1,3) & Move(n-1,1,2) & Move(1..n-2,3,2)

...

Ejemplo: Find path from a to z in graph (not shown)
```

```
Path(a->z):
    Path(a->z, f)
    Path(a->z, g)

Path(a->z, f):
    Path(a->f) & Path(f->z)

Path(a->z, g):
    Path(a->g) & Path(g->z)

...

------------------------------------------------------------------------

Arboles y Grafos AND/OR:

Dos tipos de nodos, cada cual representa una tarea a realizar
pero dos descomposiciones distintas:

Node OR: representa una tarea que para realizarla se necesita
primero realizar una de las tareas correspondientes a sus hijos.
Este nodo representa una eleccion.

Nodo AND: representa una tarea que para realizarla se necesita
primero realizar todas las tareas correspondientes a sus hijos.
Este nodo no representa una eleccion.

Hojas (terminales): tareas atomicas que no se subdividen.

Ejemplos: Torres de Hanoi, Find path.

------------------------------------------------------------------------

Solucion de Arbol AND/OR T: subarbol tal que:

1) tiene la raiz de T

2) si tiene node OR X de T, tiene uno solo hijo de X (la subtarea
   seleccionada)

3) si tiene nodo AND X de T, tiene todos los hijos de X.

Para el costo de la solucion, pueden existir varios criterios:

1) Suma de todos los costos de las aristas en T
2) Costo del camino de mayor costo en T
3) Costo Promedio sobre caminos
4) etc.

------------------------------------------------------------------------

Algoritmo AO*:

Algoritmo de busqueda heuristica tipo best-first que mantiene la
mejor solucion parcial hasta un momento dado. Asume que el grafo
AND/OR es implicito y mantienen una representacion explicita parcial
del grafo implicito. La parte explicita se le llama la parte
"explicada" del grafo implicito.

Pseudocode for AO*

========================================================================
VII. Busqueda no-deterministica. Algoritmos.
========================================================================

Problemas de busqueda no-deterministicos. Modelo dado por:

1. Conunto finito de estados S
2. Estado inicial s0 en S
```



```
3. Estados objetivos S_G
4. Acciones A(s) applicables en c/ estado
5. Funcion de transicion no deterministica F(s,a)
6. Costos c(s,a)

Solucion: estrategia para llevar el estado s0 a un estado goal.
No es una secuencia de acciones por las transiciones no-deterministas.
Se necesita consideras acciones que son funciones de estados en
acciones, llamadas politicas.

Ejemplo: navigation in grid w/ non-det actions. Depict policy in
grid.

For policy pi, define its executions. Then, pi is strong solution
or strong cyclic solution in terms of executions. Strong solution
if all its executions end in a goal state. Strong cyclic if every
execution that do not end in a goal state is unfair. An execution
is unfair if there is a state s that appears infinitely often
in the execution and there is an outcome s' in F(s,pi(s)) that
only appears a finite number of times in execution.
Strong solutions can be assigned costs but not for strong cyclic
solutions.

For policy pi, define equations for cost of pi (plain and optimistic).
If pi is strong solution, then V^pi(s0)<infty.  If pi is strong-cylcic
solution, V_min^pi(s) < infty for every state s reachable from s0
using pi. This will help us to design algorithms. Likewise, define
Bellman equations.

Problem can be mapped into an AND/OR graph (that may contain cycles).
OR nodes for states s in S, and AND nodes (s,a) for s in S and
a in A(s). Also, we can consider AND/OR graph of histories with
OR nodes (s0,s1,..,sn) and AND nodes (s0,...,sn,a) for a in A(sn).
Graph is acyclic and possibly infinite. However, if there is
a strong solution, AO* finds it in finite time.

Alternatively, can solve Bellman equation and compute greedy policy.

For computing strong cyclic solutions, can use the optimistic
Bellman equation. First, let S' = S and solve V_min for every
state in S'. Then, remove from S' every s such that V_min(s)=infty
and actions a in A(s) that lead to removed s'. Iterate until
fix point is reached. If s0 is removed, then problem has no
strong solution. Else, the policy greedy wrt to last value
function is strong solution.

Finally, one can utilize more complex algorithms for cyclic
AND/OR graphs that find best policy wrt expected cost, but
this requires that the transition function F(.,.) to be
extended w/ transition probabilities. More about this later.

========================================================================
VIII. Arboles de Juego. Valor. Minmax y alpha-beta pruning.
========================================================================

Modelo de arbol de juegos: juegos de 2 personas y suma cero,
con informacion completa. Two types of nodes: MAX and MIN.
In big games, the tree is pruned to certain level that may
not be regular across the whole tree. The leaves are assigned
a heuristic evaluation function that measures the merit of
the board configuration represented by the node. Then, an
algorithm that treat such values as the value for the game
is applied to try to find a best movement.

The tree can be searched with minimax algorithm. Observing
that max(a,b) = -min(-a,-b). Minmax can be expressed as
follows which is known as Negamax:

  minimax(node, depth)
```

```
    if node is a terminal or depth <= 0:
      return heuristic(node)
    aplha := −infty
    for child in node:
        alpha := max(alpha, −minimax(child, depth−1))
    return alpha

[Run minimax in example "AB_pruning.svg"] Show principal
variation of the game, and that any deviation by one of
the players from the PV is a suboptimal move.

Complexity of Minimax: O(b^d) but maybe far less if solution
for subtrees are cached and reused.

Alpha−Beta Pruning
------------------
Alpha−beta pruning is an algorithm that seeks to decrease the number
of nodes that are evaluated by the minimax algorithm. It stops
completely evaluating a move when at least one possibility has been
found that proves the move to be worse than a previously examined
move. Such moves need not be evaluated further. When applied to a
standard minimax tree, it returns the same move as minimax would,
but prunes away branches that cannot possibly influence the final
decision.

Diagram w/ motivation:

       [max]
      /     \
     /       \
  (min)     (min)
  alpha     /   \
           /     \
          beta   [pruned: if beta <= alpha]

This is a shallow cut−off but there may be other type of cut−offs,
called deep cut−off.

  alphabeta(node, depth, alpha, beta, Player)
    if depth = 0 or node is a terminal
        return heuristic(node)
    if Player = MaxPlayer
      for each child of node
        alpha := max(alpha, alphabeta(child, depth−1, alpha, beta, not(Player)))
        if alpha >= beta
          break                             (* Beta cut−off *)
      return alpha
    else
      for each child of node
        beta := min(beta, alphabeta(child, depth−1, alpha, beta, not(Player)))
        if alpha >= beta
          break                             (* Alpha cut−off *)
      return beta

  (* Initial call *)
  alphabeta(origin, depth, −infinity, +infinity, MaxPlayer)

With an (average or constant) branching factor of b, and a search
depth of d plies, the maximum number of leaf node positions evaluated
(when the move ordering is pessimal) is O(b*b*...*b) = O(b^d) â€“^@M–^S the
same as a simple minimax search. If the move ordering for the search
is optimal (meaning the best moves are always searched first), the
number of leaf node positions evaluated is about O(b*1*b*1*...*b)
for odd depth and O(b*1*b*1*...*1) for even depth, or O(b^{d/2}) =
O(\sqrt{b^d}). In the latter case, where the ply of a search is even,
the effective branching factor is reduced to its square root, or,
equivalently, the search can go twice as deep with the same amount of
computation. The explanation of b*1*b*1*... is that all the first player's
moves must be studied to find the best one, but for each, only the best
```



```
second player's move is needed to refute all but the first (and best)
first player move â€“^@M–^S alpha−beta ensures no other second player moves
need be considered. When nodes are ordered at random, the average number
of nodes evaluated is roughly O(b^{3d/4}).

=====================================================================
IX. Juegos. Algoritmos: Scout y Negascout.
=====================================================================

Scout

Alpha−beta pruning seems the best that can be done. Can we do better?

Intuition: when in a Max node, suppose that we know a way for
obtaining 300 points at the first branch. If there is an efficient
way to know that the second branch yields at most 300 points,
there is no need to search the second branch in *detail*.

It may be possible to verify whether the value of a branch is
greater than a given value in a way that is faster than computing
its exact value.

High−level idea:

− While searching a branch Tb of a Max node, if we have already
  obtained a lower bound of v:

  First TEST whether it is possible for Tb to return something
  better than v. If false, there is no need to search Tb, called
  fail the test. If true, search Tb, called passes the test.

− While searching a branch Tb of a Min node, if we have already
  obtained an upper bound v:

  First TEST whether it is possible for Tb to return something
  better than v. If false, there is no need to search Tb, called
  fail the test. If true, search Tb, called passes the test.

  Test(node n, depth d, value v, condition >)
  if d == 0 or terminal(n) then return TRUE if h(n) > v, else FALSE
  for n' in Succ(n) do
    if n is Max and Test(n', d−1, v, >) is TRUE, return TRUE
    if n is Min and Test(n', d−1, v, >) is FALSE, return FALSE
  if n is Max, return FALSE, else return TRUE

  Algorithm Test can be adapted to change condition to >=, <, <=

  Scout(n, depth d)
  if d == 0 or terminal(n) then return h(n)
  Let n_1, n_2, ..., n_m be the successor of n
  v := Scout(n_1, d−1)
  for i = 2 to m do
    if n is Max and Test(n_i, v, >) then v := Scout(n_i)
    if n is Min and Test(n_i, v, >=) then v := Scout(n_i)
  return v

Discussion of Scout

Test may visit less nodes than alpha−beta pruning [example]
Scout may visit a node that is prune by alpha−beta

For Test to return TRUE at subtree T, it needs to evaluate
at least: one child for a MAX node in T, all children for
a MIN node in T. If T has fixed bf and uniform depth, the
number of evaluated nodes is at least b^{d/2}

For Test to return FALSE at subtree T, it needs to evaluate
at least: one child for a Min node in T, all children for
a Max node in T. If T has fixed bf and uniform depth, the
```

```
number of evaluated nodes is at least b^{d/2}

A node may be visited more than once: one visit due to Test
and other due to Scout.

Scout show great improvements for deep games with small
branching factor. May be bad for games with large branching.

------------------------------------------------------------------------
In an alpha-beta with window [alpha,beta]:

- Failed high means it return a value larger than beta
- Failed low means it return a value smaller than alpha
- Null or zero window is a window [m,m+1].
  The result can be either:

  - Failed-high or m+1 meaning that the value is at >= m+1.
    Equivalent to Test(n,m,>) is true.

  - Failed-low or m meaning that the value is at <= m.
    Equivalent to Test(n,m,>) is false.

Alpha-beta + Scout

Intuition: try to put Scout and alpha-beta together. Search with
null windows can be used as Test in Scout. Can also apply alpha-beta
cutoffs if apply.

  [[GET SOURCE FOR NEGASCOUT FROM SLIDES]]

Negascout = Negamax + Scout

  [[GET SOURCE FOR NEGASCOUT FROM SLIDES]]

========================================================================
X. Planificacion.
========================================================================

Planificacion es "model-based approach to autonomous behaviour".
A partir de una descripcion del problema que se quiere resolverm
en donde se especifica en un lenguaje de alto nivel, la situacion
initial, las situaciones objetivos, y las acciones aplicables,
un planificador debe encontrar una solucion, llamada plan, al
problema. El planificador es entonces un solucionador general de
problemas para una clase en particular de problems, aquellos que
pueden ser expresados en el lenguaje.

    --- description ---> [PLANNER] --- solution --->

Planificacion clasica consiste en considerar problemas deterministicos
con informacion completa. Existen varios lenguajes de descripcion de
problemas, entre ellos STRIPS, ADL y SAS.

En STRIPS y ADL, una configuracion del mundo se codifica como una
valuacion logica sobre un conjunto de proposiciones. Tipicamente,
un estado s se describe como el conjunto de las proposiciones que
son asignadas por s el valor de verdad TRUE.

Ejemplo: Blocksworld: Descripcion de estados.

Los operadores en STRIPS corresponden a tripletas a=<P,A,D> de
conjuntos de proposiciones.

[[SEMANTICA DE STRIPS]]

Ejemplo: Blocksworld: Operadores.

ADL extiende a STRIPS permitiendo: precondiciones negativas,
disjunciones en las precondiciones y efectos condicionales.
```



```
Basta considerar precondiciones negativas y efectos condicionales.

Compilar ADL into STRIPS: get rid of negative preconditions,
and conditional effects.

SAS: lenguaje como generalizacion de STRIPS.

[[Algo de COMPLEJIDAD]]

========================================================================
XI. Planificacion: Heuristicas.
========================================================================

Problemas de planificacion resueltos con algoritmos de
busqueda, que necesitan heuristicas. Las heuristicas
deben calcularse a partir de la descripcion del problema
y deben funcionar para cualquier problema STRIPS.

Relajacion: h+. Propiedades importantes:

1. Basta applicar una accion a lo sumo una vez
2. Planes cortos
3. Scheduling is polynomial
4. Decomposability of plans, exploited by heuristics

Heuristicas basicas: Additive y Max.

Equations based on states:

h(s) = h_add(G;s)
h_add(p;s) = 0 if p in s; min_{a \in O(p)} c(a) + h_add(Pre(a);s)
h_add(Pre(a);s) = \sum_{q\in Pre(a)} h_add(q;s)

Can be computed in linear time. Max version is admissible
and defined by the equation that results of replacing sum
with max.

[[GIVE PSEUDOCODE FOR h_add y h_max]]

------------------------------------------------------------------------

Relaxed plan heuristic.

Problem w/ h_add: double counting of actions, do not take into
account side effects of actions. Can be fixed with Relaxed PG.

Layered fluent and action levels.

  P_0 = s
  A_0 = { a : Pre(a) \subseteq P_0 }
  P_1 = P_0 \cup { q \in Add(a) | a \in A_0 }
  ...
  A_k = { a : Pre(a) \subseteq P_k }
  P_{k+1} = P_k \cup { q \in Add(a) | a \in A_k }
  ...

Polynomial computation as there are at most |F| layers.
From this, extract relaxed plan for G, starting from the
first layer that contains G. If no such layer, problem
is not solvable. For each subgoal p at layer k, find
action at previous layer that add p and add preconditions
of p as subgoals. Once the relaxed plan is computed,
heuristic is number of action in it.

[[GIVE PSEUDOCODE FOR h_FF]]

Relaxed plan can be obtained w/ best supporters for h_max. Can
be improved by considering best supporters for h_add.
```

```
======================================================================
XI. CSPs.
======================================================================

Problem: assignment problem. Defined by variables, domains and
constraints. Task: find assignment that satisfy constraints.
Formally, a CSP is made of variables X_i, domains D_i for each
variable X_i, constraints C_j. Each constraint C_j involves some
subset of variables and specifies the *allowable* combinations
of values.

A state is a (partial) assignment of values to variables. If it
does not violate any constraint, the assignment is consistent.
A consistent state that mention all variables is a solution.

Constraint Graph: Undirected graph. Nodes represent variables.
Edge (X,Y) means that there is a constraint that mention both
X and Y.

CSP can be formulated as a search problem, where

* Initial state is empty assignment
* Successor function extends a partial assignment with X=x for
    a variable X not mentioned in the assigment, provided that
    the resulting assignment is consistent.
* Goal states are complete assignments that, by previous case,
    are consistent
* Uniform costs equal to 1

Alternative formulation: states are complete assignments, not
necessarily complete, and edges connect assignments (e.g.,
two assignments that differ in the value of just one variable).
This formulation used by local search methods.

Example: 8-Queens, Sudoku, Boolean CSPs (made of boolean variables),
  cryptoarithmetic (SEND + MORE = MONEY), etc.

Type of constraints: unary, binary, higher order. Higher-order
constraints can be converted into binary constraints by adding
additional variables.

Naive branching: branching factor at top level is nd, at second
level (n-1)d, etc. Total number of leaves is then n! * d^n, yet
the different number of assignments is only d^n. Hence, there
are a lot of duplicates and this branching is not good.

Problem: commutativity of assignments: X=1,Y=2 is equiv to Y=2,X=1.

Smart Branching: Given a node, the branches correspond to the
different values for a fixed unassigned variable. The branching
factor is thus only d. Since there are n variables, the number
of leaves is d^n that equal the number of assignments.

Can be solved w/ simple depth-first backtracking algorithm:

  Recursive-Backtracking(A, CSP)
    if A is complete, return A
    var := select-unassigned-variable(A, CSP)
    for each value in D_var do
        if var = value is consistent w/ A wrt CSP, then
            A' := A union { var = value }
            result := Recursive-Backtracking(A', CSP)
            if result != FAIL, return result
        end if
    end for
    return FAIL

Example: Australia map given by nodes {WA, NT, SA, Q, NSW, V, T}
and edges:
```



```
{ (WA,NT), (WA,SA), (NT,SA), (NT,Q), (SA,Q), (SA,NSW), (SA,V),
  (Q,NSW), (NSW,V) }

Example of backtraking:

Critical issues when implementing solution:

1. Which variable should be chosen? How should its values be ordered?

2. What are the implications of current assignment for other unassigned
   variables?

3. When a path fails, can the search avoid repeating the failure in
   subsequent paths?

----------------------------------------------------------------------

Variable and Value ordering

Idea: choose the most constrained variable in order to detect a
failure as soon as possible, because it is better to fail high
on the tree than deep into it. Heuristic is called MRV (Minimum
Remaining Values), Most Constrained Variable, or "fail-first".

Also, degree heuristics. Choose variable involved in most constraints.
It can be used as a tie-breaker when using MRV.

Once variable is selected, its values must be ordered. Least-constraining
value is an effective heuristic. It prefers values that rule out the
fewest choices for the neighboring variables. Once the variable is
fixed, try a value that will max the chances of finding a solution.

======================================================================
XI. CSPs.
======================================================================

Propagating Information through Constraints

Search space can be reduced by looking into the domain of variables
before they are selected. This techniques can be implemented even
before the search starts.


Forward Checking
----------------

Whenever a variable X is assigned, the FC process look at each unassigned
variable Y that is connected to X by a constraint, and deletes from Y's
domain any value that is inconsistent with the value chosen for X. Partner
of the MRV heuristic: select the variable w/ current smallest domain.

Example 1:
[[Table de FC on Australia Map: order SA=R, .. (choose vars using MRV)]]
                WA   NT    Q   NSW    V   SA    T
Initial        RGB  RGB  RGB  RGB  RGB  RGB  RGB
After SA=R      GB   GB   GB   GB   GB  [R]  RGB
After V=G       GB   GB   GB    B  [G]  [R]  RGB
After NSW=B     GB   GB    G  [B]  [G]  [R]  RGB
After Q=G       GB    B  [G]  [B]  [G]  [R]  RGB
After NT=B       G  [B]  [G]  [B]  [G]  [R]  RGB
After WA=G     [G]  [B]  [G]  [B]  [G]  [R]  RGB
After T=R      [G]  [B]  [G]  [B]  [G]  [R]  [R]

Example 2:
[[Table de FC on Australia Map: order WA=R,Q=G,V=B]]
                WA   NT    Q   NSW    V   SA    T
Initial        RGB  RGB  RGB  RGB  RGB  RGB  RGB
After WA=R     [R]   GB  RGB  RGB  RGB   GB  RGB
```

```
After Q=G       [R]    B  [G]  R B  RGB    B  RGB
After V=B       [R]    B  [G]  R    [B]   --  RGB
Failure

In Example 2, FC is not able to detect an inconsistency right after
the assignment Q=G, when the domains of the two connected vars
NT and SA become equal to {B}. The reason for this is that FC only
looks into neighbouring variables and not deep in the graph. This
is solved with Arc Consistency.


Arc Consistency
---------------

  AC-3(CSP)
  ---------
  while Q is not empty do
      (X,Y) := Q.first()
      if Remove-Inconsistent-Values(X,Y) then
          for each Z in Neighbors[X] do
              Q.add((Z,X))
          end for
      end if
  end while

  Remove-Inconsistent-Values(X,Y)
  -------------------------------
  removed := false
  for each x in Domain[X] do
      if no value y in Domain[Y] satisfies the constraint between X and Y then
          Remove x from Domain[X]
          removed := true
      end if
  end for
  return removed

  Time = O(n^2d^3) by bounding #insertions in queue

Higher-order Consistency
------------------------

Path consistency: triplets of variables
k-consistency: involving sets of k variables. Time = k-degree polynomial

CSP is strongly k-consistent iff it is j-consistent for j=1,..,k

Consistency can be enforced once, before search starts, or continuosly,
each time that an assignment is done.

Enforcing k-consistency requires time O(n^k)

Algorithm: enforce n-consistency and solve it backtrack free. Explain


Tree Structure
--------------

If constraint graph is tree, CSP can be solved in quadratic time (actually
linear time):

1) Choose any variable as the root and order variables from the root
to the leaves so that every node's parent in the tree precede it in
the ordering.

2) Enforce arc consistency in all nodes (it can be done better by enforcing
directional arc consistency over selected edge directions, those appearing
in the ordering computed in 1).

3) Assign values from first to last variable in the order backtrack free.
```



```


General Graph Structure
-----------------------

Two approaches:

  -- cutset conditionioning: find cutset that make graph tree and search
     over all intantiations of cutset one that generates solution

  -- find an elimination order and enforce directional k-consistency along
     the order. Then construct solution in a backtrack-free manner. The
     k that is needed is equal to the width of elimination order which
     is related to the treewidth of the graph. Explain treewidth.


Example on Australia: solve in two ways.


Backtracking: REVISE THIS
-------------

Consider the ordering and assignment {Q=R, NSW=G, V=B, T=R } ....

Conflict set for variable X is the set of previously assigned
variables that are connected to X.

Backjumping = backtrack to the most recent variable in the conflict set.

FC can supply the conflict set without extra work. Whenever FC based
on an assigment to X deletes a value from Y's domain, it should add
X to Y's conflict set. Also, every time the last value is deleted
from Y's domain, the variables in the conflict set of Y are added
to the conflict set of X. Then, when we get to Y, we know immediately
where to backtrack if needed.

Summary:

Let Xj be current variable, and conf(X_j) its conflict set. If every
possible value of Xj fails, backjump to the most recent variable Xi
in conf(X_j) and set, Conf(Xi) := Conf(Xi) union Conf(Xj) - {Xi}.

Conflict-directed backtracking makes the right jump in the recursion,
but does not avoid making the same mistake later in the search.
Constraint learning adds new constraints that are induced by the
discovered conflicts.
```

```
======================================================================
XII. SAT.
======================================================================

[SAT-CNF. CNF vs. DNF. 2-CNF. Forma clausal. Condicionamiento.
Condicionamiento en forma clausal.]

Running example:

Delta = {
  1. {A,B}
  2. {B,C}
  3. {-A,-X,Y}
  4. {-A,X,Z}
  5. {-A,-Y,Z}
  6. {-Z,X,-Z}
  7. {-A,-Y,-Z} }


Resolution: explain propositional resolution (this also provides base
for the later first-order resolution). This is a rule of inference.
This rule of inference is complete for propositional logic. Define
resolution closure.

Unit resolution: explain unit resolution, its complexity and how it
can be implemented efficiently. This is also a rule of inference but
it is not complete. It is a limited form of inference. Define the
unit-resolution closure of a theory.

Solving SAT: three approaches:
  -- pure inference
  -- pure search
  -- search + inference


Pure inference:

Compute the resolution closure of theory, and then construct solution
in a backtrack-free manner following an ordering of the variables.

Pure search:

Algoritmo SAT-I: backtracking simple, no early backtrack.
Algoritmo SAT-II: backtracking, early detection of failure or success
  (if conditioned theory becomes empty, then success. if conditioned
  theory contains empty clause, then failure), literal choosing.
  Similitud w/ CSPs.

Search + inference:

1st approach:

Algoritmo DPLL: w/o conditioning by adding units to theory.

DPLL(Delta)
-----------
1. If (I,Gamma) := Unit-Propagation(Delta)
2. If Gamma = {} then return I
3. If Gamma contains {} then return FAIL
4. L := choose literal in Gamma
5. If M := DPLL(Gamma union {{L}}) != FAIL then return M union I
6. If M := DPLL(Gamma union {{-L}}) != FAIL then return M union I
7. return FAIL

Define termination tree and show it over example with var ordering
[A,B,C,X] and always setting true before false]

======================================================================
XIII. SAT.
```



```
======================================================================

DPLL(Delta)
-----------
1. If (I,Gamma) := Unit-Propagation(Delta)
2. If Gamma = {} then return I
3. If Gamma contains {} then return FAIL
4. L := choose literal in Gamma
5. If M := DPLL(Gamma union {{L}}) != FAIL then return M union I
6. If M := DPLL(Gamma union {{-L}}) != FAIL then return M union I
7. return FAIL

----------------------------------------------------------------------
Example

Delta = {
  1. {A,B}
  2. {B,C}
  3. {-A,-X,Y}
  4. {-A,X,Z}
  5. {-A,-Y,Z}
  6. {-Z,X,-Z}
  7. {-A,-Y,-Z} }

[Show termination tree of DPLL w/ order [A,B,C,X], always setting
true before false]

The theory Delta|A,B,C,X is false because the set of clauses
{-X,Y}, {X,Z}, {-Y,Z}, {X,-Z}, {-Y,-Z} is inconsistent. However,
DPLL does not notice this and exhausts the subtree for assignment
A=false.

Implication graphs: Two examples

Graph on literals constructed by adding decisions and unit propagation.

Every cut in the implication graph that separates decision variables
from the contradiction makes up a *conflict set*.
In example 1, we have the conflict sets {A,X}, {A,Y}, {A,Y,Z}.

Conflict set are used to analize reasons of failure, compute
backtrack level, and learn clauses. Since there are multiple
conflict sets, the questions are what is the most usefule and
how to use it.

IDEA: Choose cut where all nodes at the current level are on
one side, except for the decision node at the current level
together iwh all other nodes on the other side. Formally,

C(n) = {n} if Pa(n) = ePa(n) = {}
C(n) = ePa(n) union Union_{m \in Pa(n)} C(m) otherwise,

where Pa(n) are parents of node n which are set at the same
level as n, and ePa(n) are parents of n that are set at
earlier levels. In the example 1, C(n) = {A,X}.

C(n) defines a clause {-A,-X} that is *implied* by Delta.
The conflict set also defines the backtrack level:

bl of C(n) is the highest level of any literal in C(n)
al of C(n) is the second highest level of any level in C(n)

In principle, it is enough to backtrack to level bl, undoing
all decisions in levels bl,bl+1,bl+2,...  However, SAT solvers
backtrack to level al+1, undoing all the decisions al+1,al+2,...,
*add* the conflict clausle generated by the conflict set, and
run UP. As a result, the conflict clause becomes unit given
a new assignment.
```

```
------------------------------------------------------------------------
DPLL w/ clause learning

Need primitives:

decide(V=v) that returns a boolean, but has side effects:

1. Sets variable V to value v
2. Marks variable V as a decision variable and sets its decision
   level to the current decision level
3. Increment current decision level
4. Applies unit resolution, adding any implied variable setting to
   implication graph and assigning implication level. The implication
   level of a variable setting is the maximum of the levels assigned
   to its parents in the implication graph.

If no contradiction is achived by UP, decide(V=v) returns true.
Otherwise, decide(V=v) returns false, but after constructing a
conflict-driven clause and computed the assertion level al.

The function undo-decide(V=v) erases the decision V=v and all
implied setting that were derived from it by UP. Updates the
implication graph accordingly and decrement the current decision
level.

at-assertion-level+1() returns true if the current decision level
is equal to al+1, where al was the assertion level computed by
decide(V=V).

assert-cdc() adds the conflict driven clause to KB and applied
UP. If no contradiction appears during UP, the function returns
true. Otherwise, returns false after constructing another
conflict-driven clause and computing a new assertion level.

Boundary conditions for new algorithm:

1. Initially, current decision level is 0
2. Conflict-driven clause may be a unit clause. In such case,
   al is -1 implying a backtrack to level 0 (undoing all decisions
   made).
3. Initially, KB has no unit clauses. Can be easily enforced in
   linear time by running UP on the initial KB.

The algorithm DPLL+
-------------------

Let \V be the set of variables which has not value yet. This is
a global variable. Initially, by 4, it contains all variables
in KB.

DPLL+(Delta)
------------
1. If \V is empty return true
2. Choose variable V and value v
3. If decide(V=v) and DPLL+() then
4.     undo-decide(V=v)
5.     return true
6. undo-decide(V=v)
7. If at-assertion-level+1() then
8.     return assert-cdc() and DPLL+()
9. return false

Example:

Delta = {
  1. {A,B}
  2. {B,C}
  3. {-A,-X,Y}
  4. {-A,X,Z}
```



```
  5. {-A,-Y,Z}
  6. {-A,X,-Z}
  7. {-A,-Y,-Z} }

First: decide(A=true), decide(B=true), decide(C=true), decide(X=true)
Then, UP finds a contradiction and derives the clause {-A,-X}. The
assertion level is 0 and thus backtrack occurs to level 1, undoing
decisions decide(X=true), decide(C=true), decide(B=true). The clause
{-A,-X} is added and UP is run. UP finds a new contradiction and
computes the clause {-A}. The assertion level is now -1 and then,
the algorithm backtracks to level 0 undoing all decisions. The
new clause is added and UP is run again. This time, UP set A=false
and B=true, and Delta is satisfied.

========================================================================
XIV. Resolucion.
========================================================================

Recordar sintaxis y semantica de LP; nociones de sat, tautologia,
unsat. Regla de inferencia de resolucion. Ejemplo de resoluciones,
y ejemplo de refutacion. Principio de refutacion: probar F -> G
es lo mismo que probar F & -G es UNSAT, y esto ultimo se prueba
buscando una refutacion. Algoritmo general de resolucion: buscar
refutacion. Pseudocodigo. Algoritmo es correct y completo. Termina
cuando no existen nuevas clausulas derivadas cuyo numero maximo
es finito.

Sintaxis de LPO. Semantica (solo para sentencias): definicion de
estructura. Definicion intuitiva (con ejemplo) de semantica.
Nociones: SAT, TAUT, UNSAT, Consecuencia Logica.

========================================================================
XV. Resolucion en LPO.
========================================================================

Example:

EyAz[ P(z,y) <=> -Ex[P(z,x) & P(x,z)] ]

Clausal form:

C1 = { -P(z1,a), -P(z1,x), -P(x,z1) }
C2 = { P(z2,f(z2)), P(z2,a) }
C3 = { P(f(z3),z3), P(z3,a) }

Ground clauses:

G1 = { -P(a,a) } <= C1[x/a,z1/a]
G2 = { P(a,f(a)), P(a,a) } <= C2[z2/a]
G3 = { P(f(a),a), P(a,a) } <= C3[z3/a]
G4 = { -P(f(a),a), -P(a,f(a)) } <= C1[z1/f(a),x/z]

There is a refutation of this. [Show it]

Thm: Ground resolution is complete for ground clauses
Pf: direct from resolution for prop. logic.

Thm: if a clausal form is inconsistent, there is
an instantiation of them into ground clauses from
which a refutation can be constructed.

Unification:

Unify(L,L') returns substitution theta such that L[theta] = L'[theta],
and theta is substitution that make least commitments.

MGU:

Unify(L,L') or Unify(T,T')
```

```
Fatal disagreements:

  1. C=R(−) & C’=P(−), or T=f(−) & T’=g(−)
  2. T=x and T’!=x and x appears in T’; e.g., T=x ; T’=f(x)

Repairable disagreements:

  1. T=x & T’!=x and x does not appear in T’. Extend unifier w/ x/T’


Examples: Unify(T,T’) where

1. T=f(x,f(x,y)) and T’=f(g(y),f(g(a),z)) => Theta = [x/g(a),y/a,z/a]

Resolution Procedure:

  while Delta does not contain empty clause do
      Choose two clauses C and C’ from Delta.
      Let L, L’ be two literals in C and C’, and theta=Unify(L,−L’).
      if theta != Fail then
          Delta := Delta union { Resolvent(C[theta],C’[theta[) }
      end if
  end while

Example of resolution proof:

Resolve(C2,C1) to find Theta=[z1/a,z2/a,z/a] and C4={ P(a,f(a)) }
Resolve(C1,C3) to find Theta=[z1/a,x/a,z3/a] and C5={ P(f(a),a) }
Resolve(C1,C5) to find Theta=[z1/f(a),x/a] and C6={ −P(a,f(a)) }
Resolve(C4,C6) to find Theta=[] and C7={ }

Thm: resolution procedure is guaranteed to terminate if Delta
is inconsistent.

Example:

1. Jack owns a dog. Ex[ Dog(x) & Owns(Jack,x) ]
2. Every dog owner is an animal lover. Ax[ Ey [ Dog(y) & Owns(x,y) ] => AnimalLo
ver(x) ]
3. No animal lover kills an animal. Ax [ AnimalLover(x) => Ay [ Animal(y) => −Ki
lls(x,y) ] ]
4. Either Jack or Curiosity killed the cat, who is named Tuna.
     Kills(Jack,Tuna) or Kills(Curiosity,Tuna)
     Cat(Tuna)
     Ax[ Cat(x) => Animal(x) ]
5. Did Curiosity kill the cat?
```