The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, AUTUMN SEMESTER 2010-2011

PLANNING AND SEARCH

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced

Answer FOUR out of SIX questions

Only silent, self contained calculators with a Single-Line Display are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn your examination paper over until instructed to do so

1. This question is on genetic algorithms.

   (a) Define the terms chromosome, fitness function, crossover and mutation as used in genetic algorithms. Explain how genetic algorithms work, in English or in pseudocode. (10 marks)

   Answer. Genetic algorithm is essentially stochastic local beam search which generates successors from *pairs* of states. It works with k states (chromosomes or individuals). This set is called population. Original population is randomly generated. Each individual represented as a string (chromosome). Each individual is rated by a maximising objective function (*fitness function*). Probability of being chosen for reproduction is directly proportional to fitness. Two parents produce offspring by *crossover*. Then with some small probablity, mutation is applied (bits of the string changed).

```
function GENETIC-ALGORITHM(population,FITNESS-FN) returns an individual
    inputs: problem, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual
    repeat
        new-population ← empty set
        for i=1 to SIZE(population) do
            x ← RANDOM-SELECTION(current,FITNESS-FN)
            y ← RANDOM-SELECTION(current,FITNESS-FN)
            child ← REPRODUCE(x, y)
            if (small random probability) then child ← MUTATE(child)
            add child to new-population
    until some individual is fit enough or enough time has elapsed
    return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE( x, y) returns an individual
    inputs: x, y, parent individuals
    n ← LENGTH(x); c ← random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))
```

   (b) A genetic algorithm is to be used to evolve a binary string of length n containing only 1s. The initial population is a randomly generated set of binary strings of length n.

      i. Give a suitable fitness function for this problem. (2 marks)

      Answer. The most obvious function would be the sum of 1s in the string.

      ii. Will the offspring of parents with a high fitness value generally also have a high fitness value, given your fitness function? Explain your answer. (2 marks)

      Answer. Yes, in general, if two strings have a high number of 1s, then their offspring is likely to have a high number of 1s too (although it is possible that the offspring will have fewer 1s, for example, 0011 and 1100 may produce 0000).

(c) Suppose the problem is to evolve a binary string of length n which is symmetric. If the string positions are numbered from 0, then a symmetric string will have a 1 in position i if and only if there is a 1 in position $(n-1)-i$. For example, 001100 is symmetric since it has a 1 at index 2 and a 1 at index $(6-1)-2=3$. Similarly, 110011 is symmetric, and 011011 is not. The initial population is a randomly generated set of binary strings of length n, where n is an even number.

i. Give a suitable fitness function for this problem. (3 marks)
Answer. A possible fitness function is $n-x$, where x is the number of unmatched 1s in both halves of the string.

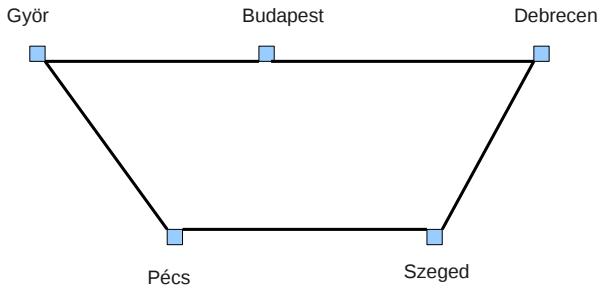
ii. Will the offspring of parents with a high fitness value generally also have a high fitness value, given your fitness function? Explain your answer. (3 marks)
Answer. For the function above, no (the probability that the result of crossover of two almost symmetric strings is almost symmetric is low).

(d) If the population size in a genetic algorithm is restricted to 1, what search algorithm does it correspond to? Explain your answer. (5 marks)

Answer. If the population is always size one, crossover applied to two copies of the same string will return the original string, so the only operation which does change the solution is random mutation. In this case, since there is no attempt to improve the solution, the algorithm degenerates to a random walk through the search space.

2. This question is on search with non-deterministic actions.

   Consider the following search problem. The set of states corresponds to a set of cities on the map of Hungary below (the state Budapest corresponds to being in Budapest, etc.). Actions correspond to following an edge (driving along the road) from one city to another. All actions apart from two are deterministic and have the expected result, namely driving from X to Y results in being in Y. The only exceptions are actions which involve driving out of Győr: the action of driving from Győr to Budapest has two possible outcomes, one being in Budapest and another being in Pécs. Similarly, the action of driving from Győr to Pécs also has two possible outcomes, one being in Budapest and another being in Pécs. The initial state is being in Győr and the goal is to reach Szeged.



   (a) For each state, give a list of possible actions. (2 marks)

   Answer. I will abbreviate cities by their first letter.
   $Actions(G) = \{drive(G, B), drive(G, P)\}$
   $Actions(B) = \{drive(B, G), drive(B, D)\}$
   $Actions(D) = \{drive(D, B), drive(D, S)\}$
   $Actions(S) = \{drive(S, D), drive(S, P)\}$
   $Actions(P) = \{drive(P, S), drive(P, G)\}$

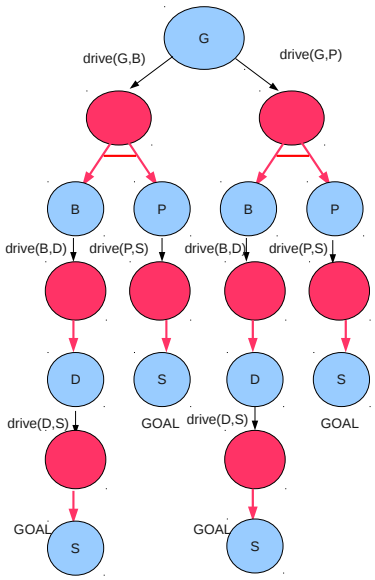
   (b) For each state and action, define the function $Result(state, action)$. (2 marks)

   Answer.
   $Result(X, drive(X, Y)) = Y$ for all X, Y apart from X=Győr.
   $Result(G, drive(G, B)) = \{B, P\}$.
   $Result(G, drive(G, P)) = \{B, P\}$.

   (c) Draw the and-or search tree for this search problem. To reduce the size of the tree, don't expand the nodes for the states you have already visited. (7 marks)

   Answer.

   (d) Explain how an and-or search algorithm works, in English or pseudocode. (10 marks)



Answer. And-or search takes as an input an and-or search tree, where OR nodes correspond to the choice of actions, and AND nodes correspond to environment's responses.

A solution for an AND-OR search problem is a subtree that

(1) has a goal node at every leaf

(2) specifies one action at each of its OR nodes

(3) includes every outcome branch at each of its AND nodes

The following algorithm finds a non-cyclic solution if it exists. It chooses one branch at the or-nodes and searches for a solution for each branch of an and-node.

function AND-OR-GRAPH-SEARCH(*problem*) **returns** a conditional plan, or failure
OR-SEARCH(*problem*.INITIAL-STATE,*problem*,[])

---

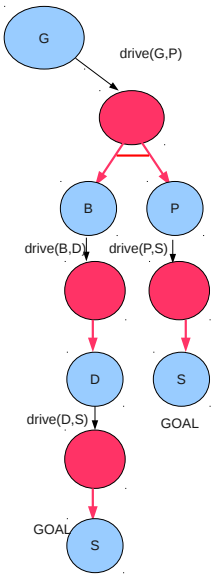
function OR-SEARCH(*state, problem, path*) **returns** a conditional plan, or failure
if *problem*.GOAL-TEST(*state*) **then return** the empty plan
if *state* is on *path* **then return** failure
for each *action* **in** *problem*.ACTIONS(*state*) **do**
plan ← AND-SEARCH(RESULTS(*state, action*), *problem*,[*state* | *path*])
if *plan* ≠ *failure* **then return** [*action* | *plan*]
return *failure*

---

function AND-SEARCH(*states, problem, path*) **returns** a conditional plan, or failure
for each s_i in *states* **do**
$plan_i$ ← OR-SEARCH(s_i,*problem, path*)
if *plan* = *failure* **then return** *failure*
return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** ... **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

(e) Draw a subtree which is a solution to the problem and state the corresponding plan. (4 marks)

Answer. Either subtree will do, here is the one which starts with the choice of driving to Pécs. The corresponding plan is [drive(G,P), if(P) then drive(P,S) else drive(B,D), drive (D,S)].



3. This question is on SATPlan algorithm.

   Consider the following planning problem:

   Initial state: $Have(Cake)$

   Goal: $Have(Cake) \wedge Eaten(Cake)$

   $Eat(Cake)$:
   PRECOND: $Have(Cake)$
   EFFECT: $\neg Have(Cake) \wedge Eaten(Cake)$

   $Bake(Cake)$:
   PRECOND: $\neg Have(Cake)$
   EFFECT: $Have(Cake)$

   (a) Give a description of this planning problem in terms of propositional formulas, suitable as an input to the SATPlan algorithm when searching for a plan consisting of 2 actions. You need to state that:

   i. The initial state description holds at time 0. (2 marks)
   Answer. $Have(Cake)^0 \wedge \neg Eaten(Cake)^0$.

   ii. The goal description holds at time 2. (1 marks)
   Answer. $Have(Cake)^2 \wedge Eaten(Cake)^2$.

   iii. Successor state axioms for times 1 and 2 for fluents $Have(Cake)$ and $Eaten(Cake)$. (8 marks)
   Answer.
   $Have(Cake)^1 \Leftrightarrow (Bake(Cake)^0 \vee (Have(Cake)^0 \wedge \neg Eat(Cake)^0))$
   $Have(Cake)^2 \Leftrightarrow (Bake(Cake)^1 \vee (Have(Cake)^1 \wedge \neg Eat(Cake)^1))$
   $Eaten(Cake)^1 \Leftrightarrow (Eat(Cake)^0 \vee Eaten(Cake)^0)$
   $Eaten(Cake)^2 \Leftrightarrow (Eat(Cake)^1 \vee Eaten(Cake)^1)$

   iv. Precondition axioms for times 0 and 1 for the two actions. (2 marks)
   Answer.
   $Eat(Cake)^0 \Rightarrow Have(Cake)^0$
   $Eat(Cake)^1 \Rightarrow Have(Cake)^1$
   $Bake(Cake)^0 \Rightarrow \neg Have(Cake)^0$
   $Bake(Cake)^1 \Rightarrow \neg Have(Cake)^1$

   v. Action exclusion axioms for times 0 and 1. (2 marks)
   Answer.
   $\neg Eat(Cake)^0 \vee \neg Bake(Cake)^0$
   $\neg Eat(Cake)^1 \vee \neg Bake(Cake)^1$

   (b) Give an assignment which satisfies all the formulas above, and show that the formulas are indeed true for this assignment. (8 marks)

   Answer.
   (1) $Have(Cake)^0 = true$
   (2) $Eaten(Cake)^0 = false$
   (3) $Eat(Cake)^0 = true$

(4) $Bake(Cake)^0 = false$
(5) $Have(Cake)^1 = false$
(6) $Eaten(Cake)^1 = true$
(7) $Eat(Cake)^1 = false$
(8) $Bake(Cake)^1 = true$
(9) $Have(Cake)^2 = true$
(10) $Eaten(Cake)^2 = true$

The initial state description is true because $Have(Cake)^0 \wedge \neg Eaten(Cake)^0$ evaluates to true given (1) and (2).

The goal state description is true given (9) and (10).

Successor state axiom for $Have(Cake)^1$ is true because $Have(Cake)^1$ is false and $(Bake(Cake)^0 \vee (Have(Cake)^0 \wedge \neg Eat(Cake)^0))$ is also false because of (4) and (3).

Successor state axiom for $Have(Cake)^2$ is true because $Have(Cake)^2$ is true and $(Bake(Cake)^1 \vee (Have(Cake)^1 \wedge \neg Eat(Cake)^1))$ is true because of (8).

Successor state axiom for $Eaten(Cake)^1$ is true because $Eaten(Cake)^1$ is true and $(Eat(Cake)^0 \vee Eaten(Cake)^0)$ is true because of (3).

Successor state axiom for $Eaten(Cake)^2$ is true because $Eaten(Cake)^2$ is true and $(Eat(Cake)^1 \vee Eaten(Cake)^1)$ is true because of (6).

Similarly for precondition and exclusion axioms.

(c) Extract solution from this assignment. (2 marks)

Answer.

$Eat(Cake)^0 = true$ and $Bake(Cake)^1 = true$ mean that the plan is $Eat(Cake)$, $Bake(Cake)$.

4. (a) Explain how regression planning works, in English or pseudocode. (10 marks)

Answer. Regression planning starts at the goal state(s) and does regression (goes back towards the initial state). Given a goal description g and a ground action a, the regression from g over a gives a state description g':

$g' = (g - \textsc{Add}(a)) \cup \{\textsc{Precond}(a)\}$

The regression is done over *relevant* actions: those that have an effect which is in the set of goal elements and no effect which negates an element of the goal.

Search backwards from g, remembering the actions and checking whether we reached an expression applicable to the initial state. Regression search can use any search strategy, for example breadth-first or depth-first. Goal stack planning which uses essentially DFS was explained in greater detail in the lectures, students may just give the goal stack planning algorithm for full marks.

(b) Consider the following planning problem. An agent is at home and has no food. The goal is to be at home and have food. Actions available to the agent are:

$Buy(Food)$:
PRECOND: $At(x) \wedge Sells(x, Food)$
EFFECT: $Have(Food)$

$Go(x, y)$:
PRECOND: $At(x)$
EFFECT: $\neg At(x) \wedge At(y)$

The initial state is $At(Home) \wedge Sells(S1, Food) \wedge Sells(S2, Food)$.

The goal state is $At(Home) \wedge Have(Food)$.

Solve this problem using regression planning; trace the algorithm, explaining each step. (10 marks)

Answer.

The goal state $g = \{At(Home), Have(Food)\}$. The relevant action is $Go(x, Home)$ (strictly speaking, it is also possible to apply $Buy(Food)$, but then we will regress to a state description containing $Sells(Home, Food)$ which is impossible to make true, so we assume that there is a heuristic telling us not to try). There are two possible ground instances of this action, $Go(S1, Home)$ and $Go(S_2, Home)$. Suppose we choose to regress over $Go(S1, Home)$.

Step 1: $Go(S1, Home)$.
$Add(Go(S_1, Home)) = \{At(Home)\}$
$\textsc{Precond}(Go(S_1, Home)) = \{At(S_1)\}$.
$g_1 = (\{At(Home), Have(Food)\} - \{At(Home)\}) \cup \{At(S1)\} = \{At(S_1), Have(Food)\}$.

The relevant actions in g_1 are $Go(x, S_1)$ and $Buy(Food)$. Hopefully we have a heuristic which suggests that we should choose making $Have(Food)$ true, because it is one of the goal conjuncts, namely regress over $Buy(Food)$ rather than $Go(x, S_1)$.

Step 2: $Buy(Food)$.
$Add(Buy(Food)) = \{Have(Food)\}$
$\textsc{Precond}(Buy(Food)) = \{At(x), Sells(x, Food)\}$.
$g_2 = \{At(S_1), Sells(S_1, Food)\}$ (we can bind x to S_1).

The relevant actions in g_2 are $Go(Home, S_1)$ and $Go(S_2, S_1)$ and in this case there is definitely a heuristic to choose the first one (since the g' in this case matches the initial state). So we regress over $Go(Home, S_1)$.

Step 3: $Go(Home, S_1)$.
$Add(Go(Home, S_1)) = \{At(S_1)\}$
$\textsc{Precond}(Go(Home, S_1)) = \{At(Home)\}$.
$g_3 = \{Sells(S_1, Food), At(Home)\}$ which includes the initial state.

The resulting plan (reading backwards from g_3) is $[Go(Home, S_1), Buy(Food), Go(S1, Home)]$.

(c) How are a *classical search problem* and a *classical planning problem* formulated? Explain the difference between them. (5 marks)

Answer. A search problem is defined in terms of (1) a set of states, (2) for each state, a list of actions applicable in this state, (3) for each pair of a state and applicable action, the resulting state, (4) initial state, (5) goal state (or goal test). A planning problem is defined in terms of (1) action schemas with preconditions and effects, (2) initial state description in terms of fluents, (3) goal state description in terms of fluents. Planning problems use factored representations of states, while search problems consider states as simple atomic entities.

Marking scheme: 3 marks for just defining the problems, 2 for pointing out the most important difference (don't have to use the words factored representation).

5. This question is on partial order planning.

   (a) Explain what is a totally ordered plan and what is a partially ordered plan. (3 marks)

   Answer. A totally ordered plan is a sequence of actions (a totally ordered set of actions). A partially ordered plan is a set of actions and a set of temporal constraints $a_i \prec a_j$ (a set of actions partially ordered by the constraints).

   (b) Explain how partial order planning works (in English or pseudocode). (10 marks)

   Answer. Partial order planning searches within a space of partial plans. Partial plans always have a *Start* step which has the initial state description as its effect, and a *Finish* step which has the goal description as its precondition. They also have a list of temporal constraints $a_i \prec a_j$ stating that action a_i should precede action a_j. The algorithm also uses a list of causal links from outcome of one step to precondition of another.

   An open condition is a precondition of a step not yet causally linked. A plan is complete iff every precondition is achieved. A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it. A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. It should be either moved before the first step or after the second.

   Operators on partial plans:

   - add a link from an existing action to an open condition
   - add a step to fulfill an open condition
   - order one step wrt another to remove possible conflicts

   Gradually move from incomplete/vague plans to complete, correct plans. Backtrack if an open condition is unachievable or if a conflict is unresolvable.

```
function POP(initial, goal, actions) returns plan

    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if SOLUTION?( plan) then return plan
        S_need, c ← SELECT-SUBGOAL( plan)
        CHOOSE-ACTION( plan, actions, S_need, c)
        RESOLVE-THREATS( plan)
    end

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

function SELECT-SUBGOAL( plan) returns S_need, c

    pick a plan step S_need from STEPS( plan)
        with a precondition c that has not been achieved
    return S_need, c
```

procedure CHOOSE-ACTION(*plan, actions,* S_{need}, *c*)

choose a step S_{add} from *actions* or STEPS(*plan*) that has *c* as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *actions* **then**
add S_{add} to STEPS(*plan*)
add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
choose either
Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
if not CONSISTENT(*plan*) **then fail**
end

(c) Solve the following blocks world problem below using partial order planning; trace the search from the initial empty plan to a complete solution, explaining each step. (12 marks)

The actions are:

$Move(b, x, y)$:
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y)$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$

$Movetotable(b, x)$:
PRECOND: $On(b, x) \wedge Clear(b)$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$

The initial state is $On(B, A) \wedge On(A, Table) \wedge On(C, Table) \wedge Clear(B) \wedge Clear(C)$. The goal is $On(C, B) \wedge On(B, Table)$.

Answer.

$Steps = \{Start, Finish\}$ where the effect of $Start$ is $On(B, A) \wedge On(A, Table) \wedge On(C, Table) \wedge Clear(B) \wedge Clear(C)$, and the $Finish$ step has $On(C, B) \wedge On(B, Table)$ as its precondition.
$Links = \{\ \}$
$Orderings = \{\ \}$
Open conditions: $On(C, B)$, $On(B, Table)$ (for the $Finish$ step).

Suppose we choose the first open condition as a selected subgoal.

An action which would make it true is $Move(C, x, B)$. We add it as a step to the partial plan, and a link from it to $Finish$:

$Steps = \{Start, Move(C, x, B), Finish\}$
$Links = \{\ Move(C, x, B) \overset{On(C,B)}{\longrightarrow} Finish\}$
$Orderings = \{Start \prec Move(C, x, B), Move(C, x, B) \prec Finish\}$
Open conditions: $On(B, Table)$ (for the $Finish$ step), $On(C, x)$, $Clear(C)$, $Clear(B)$ (for the $Move(C, x, B)$ step).

There are no clobberers.

Suppose we choose the first open condition as a selected subgoal.

An action which would make it true is $Movetotable(B, y)$. We add it as a step to the partial plan, and a link from it to $Finish$:

$Steps = \{Start, Move(C, x, B), Movetotable(B, y), Finish\}$
$Links = \{\ Move(C, x, B) \overset{On(C,B)}{\longrightarrow} Finish, Movetotable(B, y) \overset{On(B,Table)}{\longrightarrow} Finish\}$
$Orderings = \{Start \prec Move(C, x, B), Move(C, x, B) \prec Finish, Start \prec Movetotable(B, y), Movetotable(B, y) \prec Finish\}$
Open conditions: $On(C, x)$, $Clear(C)$, $Clear(B)$ (for the $Move(C, x, B)$ step), and $On(B, y)$, $Clear(B)$ (for the $Movetotable(B, y)$ step).

There is also clobbering: $Move(C, x, B)$ destroys a precondition for $Movetotable(B, y)$ ($Clear(B)$). Since we cannot put $Movetotable(B, y)$ after $Finish$, we put it before $Move(C, x, B)$:

$Orderings = \{Start \prec Move(C, x, B), Move(C, x, B) \prec Finish, Start \prec Movetotable(B, y), Movetotable(B, y) \prec Finish, Movetotable(B, y) \prec Move(C, x, B)\}$

Now we can deal with the open conditions by adding a link from $Start$ to all of them: $On(C, x)$ with $x = Table$, $Clear(C)$, $Clear(B)$, $On(B, y)$ with $y = A$ is made true by $Start$.

The resulting complete plan is as follows:

$Steps = \{Start, Move(C, Table, B), Movetotable(B, A), Finish\}$
$Links = \{\ Move(C, Table, B) \overset{On(C,B)}{\longrightarrow} Finish, Movetotable(B, A) \overset{On(B,Table)}{\longrightarrow} Finish, Start \xrightarrow{c} Move(C, Table, B), Start \xrightarrow{d} Movetotable(B, A)\}$ where $c \in \{On(C, x), Clear(C), Clear(B)\}$, $d \in \{On(B, y), Clear(B)\}$.
$Orderings = \{Start \prec Move(C, x, B), Move(C, x, B) \prec Finish, Start \prec Movetotable(B, y), Movetotable(B, y) \prec Finish, Movetotable(B, y) \prec Move(C, x, B)\}$

6. This question is on HTN planning.

   Consider the following (incomplete) planning domain description. A fence consists of 3 panels, numbered 1 to 3. The agent can be positioned in front of any of the panels. The agent being in front of panel i is represented by $At(i)$. The agent can move one panel to the left and one panel to the right. When it is in front of an unpainted panel, it can paint it. The planning domain is described by the following hierarchical task network (where the refinements for $Go(x, 1)$ action are not given):

   $Left(x)$:
   PRECOND: $At(x) \wedge x > 1$
   EFFECT: $\neg At(x) \wedge At(x - 1)$

   $Right(x)$:
   PRECOND: $At(x) \wedge x < 3$
   EFFECT: $\neg At(x) \wedge At(x + 1)$

   $Paint(x)$:
   PRECOND: $At(x) \wedge \neg Painted(x)$
   EFFECT: $Painted(x)$

   $Refinement(PaintAll,$
   PRECOND: $At(x)$
   STEPS: $[Go(x, 1), Paint(1), Right(1), Paint(2), Right(2), Paint(2), Right(2), Paint(3)])$

   $Refinement(Go(x, 1),$
   ??? ???

   (a) Add suitable refinements for the $Go(x, 1)$ high-level action. (5 marks)
   Answer. The most straightforward answer would be to write a separate implementation for each x, namely
   $Refinement(Go(1, 1),$
   PRECOND: $At(1)$,
   STEPS: [])
   $Refinement(Go(2, 1),$
   PRECOND: $At(2)$,
   STEPS: [Left(2)])
   $Refinement(Go(3, 1),$
   PRECOND: $At(3)$,
   STEPS: [Left(3), Left(2)])
   A recursive refinement is also possible,
   $Refinement(Go(x, 1),$
   PRECOND: $x = 1$
   STEPS: [])
   $Refinement(Go(x, 1),$
   PRECOND: $x > 1$
   STEPS: [Left(x), Go(x-1)])

Full marks will be awarded for either of those or any other correct answer.

(b) Explain how hierarchical search works, in English or pseudocode. (10 marks)

Answer. The following algorithm explores solutions in breadth-first manner and replaces the first high-level action in the current solution with its refinements.

```
function HIERARCHICAL-SEARCH(problem, HTN) returns a solution,
or failure
    frontier ← a FIFO queue with [Act] as the only element
    loop do
        if EMPTY?(frontier) then return failure
        plan ← POP(frontier) /* chooses the shallowest plan in frontier
*/
        hla ← the first HLA in plan, or null if none
        prefix, suffix ← the action subsequences before and after hla in
plan
        outcome ← RESULT(problem.INITIAL-STATE, prefix)
        if hla is null then /* so plan is primitive and outcome is its
result */
            if outcome satisfies problem.GOAL then return plan
            else if for each sequence in REFINEMENTS(hla, outcome,
HTN) do
            frontier ← INSERT(APPEND(prefix, sequence, suffix),frontier)
```

(c) Trace hierarchical search on the following example: the planning domain is as given above plus your refinements for the $Go(x, 1)$ action. In the initial state the agent is in front of panel 2 and the fence is not painted, and in the goal state it is painted. For simplicity, assume that the only refinement of the top-level action *Act* is *PaintAll*. If you could not do part (a), just assume that $Go(x, 1)$ is a primitive action which makes $At(1)$ true. (5 marks)

Answer. At first, the frontier contains a single plan [*Act*]. *hla* is assigned *Act*. $prefix$ and $suffix$ are empty. *outcome* is the initial state. We replace *Act* with its refinement *PaintAll*, the frontier contains [*PaintAll*]. At the next iteration, we replace *PaintAll* with its refinement and the frontier contains

$$[Go(x, 1), Paint(1), Right(1), Paint(2), Right(2), Paint(2), Right(2), Paint(3)]$$

At the next iteration, *hla* is assigned $Go(x, 1)$ and $suffix$ is

$$[Paint(1), Right(1), Paint(2), Right(2), Paint(2), Right(2), Paint(3)]$$

Depending on how many applicable refinements $Go(x, 1)$ has, all of them followed by the suffix are inserted into the frontier. Supposing there is a single refinement $Left(2)$, the frontier now contains

$$[Left(2), Paint(1), Right(1), Paint(2), Right(2), Paint(2), Right(2), Paint(3)]$$

At the next iteration, *hla* is null and $prefix$ is

$$[Left(2), Paint(1), Right(1), Paint(2), Right(2), Paint(2), Right(2), Paint(3)]$$

The outcome of applying this primitive plan in the initial state is the goal state, so this solution is returned.

(d) Compare the complexity of solving an arbitrary planning problem using forward state-space search with primitive actions, and solving it using hierarchical planning. (5 marks)

Answer. Consider a non-hierarchical forward state-space planner. Suppose the solution has d steps and there are b actions possible in each state. Then the complexity of the search is $O(b^d)$. Suppose each HLA has r refinements and k steps in a refinement. For the hierarchical planner, the complexity is $O(r^{d/k})$ (branching r, depth d/k). If r is small (at most b), $O(r^{d/k})$ is root k of non-hierarchical cost $O(b^d)$.