Computer Science 433 -Artificial Intelligence
Dr. Robert Kremer
Assignment 1: Set Based Search and Or Tree Based Search

Group 8
Todd Bennett, Sean Brown, Alex Madsen

Set Based Search

In choosing a set based search, we choose to imitate nature and genetics. We create a population of solutions to our problem, and mutate or combine them in order to breed the fittest possible solution given some resource limitation. For our *Sisyphus I* problem, this means we must find a way to quickly create arbitrary room assignments, and swap people around until time runs out. Note that this does not guarantee that different runs will produce the same solution. We explore our proposed implementation of Set Based Search below.

Search Model

For our purposes, we define our search model as per Jörg Denzinger's definition (Kremer, 2013).

$$A_{SET} = (S_{SET}, T_{SET})$$

Where A_{SET} is the search model, S_{SET} is the search state, and T_{SET} is the set of transition rules, which is a relation on T_{SET}:

$$T_{SET} : S_{SET} \times S_{SET}$$

Search States

We define the S_{SET} to be the set of all possible fact sets. In the case of the *Sisyphus I* problem (Kremer, 2013), we define a fact to be a *viable* room assigning. Similarly, we say a room assigning is *viable* if and only if it violates no hard constraints.

$$S_{SET} = 2^F$$

As such, S_{SET} is the set of all possible sets of room assignings, which directly translates into S being the power set of all room assignings.

Transition (Extension) Rules

Our transition rules define the set of legal changes between search states. We chose to implement a specific version of Set Based Search known as a *genetic algorithm*. Genetic algorithms have two defining transition rules: *mutation* and *combination*. Mutation involves changing one individual within the population. Combination involves taking two facts and combining their information in some way such that one or more offspring are produced. Using these, if we use *mutation* or *combination* to get there.

$$T_{SET} = \{(s, s') \mid \exists A \rightarrow B \in Ext \bullet (A \subseteq s) \wedge (s' = (s-A) \cup B)\}$$

$$Ext = \{A \rightarrow B \mid A, B \in F\}$$

Given the *Sisyphus I* room assignment problem, we define mutation as the moving of one person from one office to another. We define this formally below.

$$inRoom : (Person, F) \times Room$$
$$inRoom(p, A) = r \leftrightarrow p \text{ is assigned to room } r \text{ in fact } A$$

$$Mut = \begin{array}{l}\{A \rightarrow B \mid \exists p,q \in A \text{ st. } p,q \in B \wedge inRoom(p, A) = inRoom(q, B) \wedge \\ inRoom(q, A) = inRoom(p,B) \bullet p \neq \phi \bullet \\ \forall r \in A, r \in B, q \neq p, q, inRoom(r, A) = inRoom(r, B)\}\end{array}$$

Mutation is, then, a change of state such that we swap the room assignment of some person in fact A with another possibly-null person's assignment within B. All other persons remain unchanged, and both states must be viable. Note that ϕ represents the null person, a placeholder person for a vacancy within the room. Also note that inRoom is a relation and not a function because of ϕ being in every room. So as long as one value of inRoom(q, A) and inRoom(q, B) makes the above statements true, it is considered to be true.

Upon consideration and correspondence with Dr. Robert Kremer (2013), we have decided not to implement combination within our algorithm. Given the time limitation of our SBS, and the relative complexity of determining a viable solution by combination, its inclusion could actually be a detriment within our algorithm.

Search Process

Our search process defines how our search behaves from decision to decision. Formally this is written as below:

$$P_{SET} = (A_{SET}, Env_{SET}, K_{SET})$$

Where P_{SET} is the search *process*, A_{SET} is our *model*, Env_{SET} represents the search environment, and K_{SET} represents the *control* function. As we have defined our model in a previous section, we must now formally define our search *environment* and *control*.

Environment

An environment must not be something that makes sense to include within the search state, or anything which may be prone to change (Kremer, 2013). Thus, the environment is more of a choice rather than formal definition.

So let us define the search environment, Env_{SET}, to be the set of all rooms and their occupants within our problem universe. We say this because it is possible that we may have a floor added to our problem, or more rooms may become available elsewhere. As well, they may already have unrelated occupants whom our team members must share an office with and we need to evaluate this possibility. Thus, this is the logical choice for Env_{SET}.

Control Function

Genetic algorithms have a unique emphasis on arbitrary decision making in order to produce the best results. Formally, we define a control function as follows:

$$K_{SET}(s, e) = (s-A) \cup B$$

Where *s* is a particular search state and *e* is a particular search environment. We also have the following conditions.

A and B must be facts such that moving from A to B is a valid extension rule. Writen formally as:

$$A, B \in F \text{ st. } A \rightarrow B \in Ext$$

A must be a set of facts within the search state. Written formally as:

$$A \subseteq s$$

The transition $A \rightarrow B$ is the most valuable of all other available transitions. We can write this formally as follows, using f_{WERT} to define a 'worth' function and f_{SELECT} to define a selection function:

$$\forall A' \rightarrow B' \in Ext \mid A' \subseteq s \bullet f_{WERT}(A,B,e) \leq f_{WERT}(A',B',e)$$

$$A \rightarrow B = f_{SELECT}(\{A' \rightarrow B' \mid \forall A'' \rightarrow B'' \in Ext \mid A'' \subseteq s \bullet f_{WERT}(A',B',e) \leq f_{WERT}(A'', B'', e)\}, e)$$

Clearly, then, the power of our search control lies within our search f_{WERT} and f_{SELECT}. We must define and tailor them for our *Sisyphus I* problem. We define f_{WERT} and f_{SELECT} using Denzinger's definitions (Kremer, 2013) below.

Worth, defined by f_{WERT}, takes the inputs of the start and end search states as well as the current environment to produce a heuristic value. While this can be any number, we are given integer values for our *Sisyphus I* problem. Therefore, rather than working with any real number, $\Re$, we would work with integers, $I \subseteq \Re$. However, the interface definition still stands formally as follows:

$$f_{WERT}: 2^F \times 2^F \times Env \rightarrow \Re$$

The heuristic value of $A \rightarrow B$ is determined by summing up the already given *Sisyphus I* heuristic assignment penalties for each assignment in the state. Given that they are of negative value, we want to assign facts that violate any hard constraint, such as *at most two people to a room*, a value of negative infinity. We write this formally as:

$$f_{WERT}(A, B, e) = -\infty \textbf{ if } \neg isViable(B, e) \textbf{ else } \Sigma\, heuristicValue(B, e), A \rightarrow B \in Env$$

If there is a tie between two transitions, we use f_{SELECT} in order to determine which transition to follow. In a genetic algorithm, we want such choices to be as various as possible. Therefore, our f_{SELECT} will simply choose randomly in the case of a tie.

The interface for f_{SELECT}, as defined by Denzinger (Kremer, 2013) is defined formally as follows:

$$f_{SELECT} : 2^{2F \times 2F} \times Env \rightarrow 2^F \times 2^F$$

$$f_{SELECT}(C, e) = A \rightarrow B \mid C \subseteq T_{SET} \bullet A \rightarrow B \in C \bullet A \rightarrow B = randomChoice(C)$$

Using these tools, we define our K_{SET} as follows. Let $A, B \in S_{SET}$ such that B is formed through *halfKill*$(A \cup M)$, where A is the current set of mutations, and M is the set formed through k mutations of random members of A, where $k = |A|$. We formally define halfKill(S) as follows.

$$halfKill: 2^F \rightarrow 2^F$$

The function *halfKill(S)* returns a set half the size of |S| with one quarter of the population of S excluded based randomly, and another quarter killed through f_{WERT} and f_{SELECT}. This should produce a healthy population of nearly ideal individuals. The justification of the ordering of random killing versus selective killing is to produce the most various population possible. These tools will be our K_{SET}.

Search Instance

Our search instance defines the starting and ending states of our search. We define this formally as per Denzinger (Kremer, 2013):

$$Ins_{SET} = (s_{0\text{-}SET}, G_{SET})$$

where $s_{0\text{-}SET}$ represents the initial state, and G_{SET} represents the goal state.

Initial State

Formally, $s_{0\text{-}SET} \subseteq S_{SET}$. Practically, since we want the maximum amount of variety between facts within our population, we define our initial state by randomly generating *k* members through a random Otree search.

$$s_{0\text{-}SET} = randomGeneration(k)$$

$$randomGeneration: Int \rightarrow 2^F$$

Goal State

Formally, we define the goal state as $s_{GOAL\text{-}SET} \subseteq S_{SET}$, and we define the goal function formally as follows:

$$G: 2^F \rightarrow \{Success, Failure\}$$

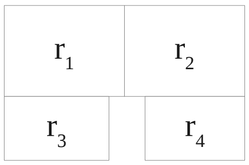
$$G(s) = s_{GOAL\text{-}SET} \subseteq s \lor \neg\, possibleTransition(s)$$

Due to the computational cost of computing $s_{GOAL\text{-}SET}$, this value will not be computed and therefore this end of *G* cannot be reached. As well, it is always possible to mutate s, meaning that this end of G will also never be reached. Thus, our only limitation on reaching our goal state is when we run out of time.

Example

This is a very simple example. The number of people ensures there is an ideal solution but also demonstrates the choices made to come to a good solution.

6 People	*4 Rooms*
1 Manager	2 Large
1 Team Lead	2 Small
2 of Manager's team	
2 of Team Lead's Team	

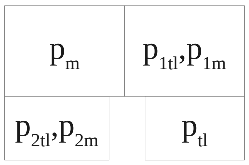


The set based search is defined by an or tree based search to start. This will find a good state to start from and thus begin mutating to find different solutions. We should also define the people and rooms in the example.

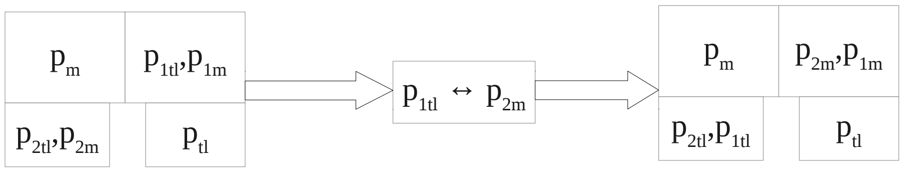
$$P = \{p_m, p_{tl}, p_{1m}, p_{2m}, p_{1tl}, p_{2tl}\}$$

$$R = \{r_1, r_2, r_3, r_4\}$$

Let us assume the or tree search brings us some arbitrary room assignments that abide by the hard constraints.

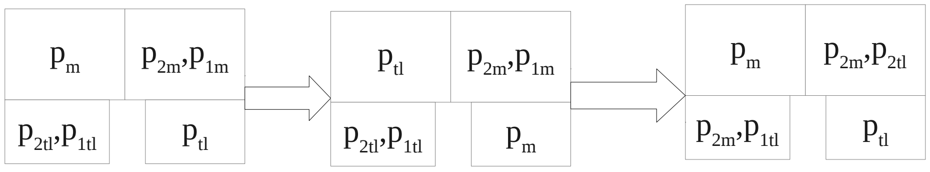


This is a very unfit assignment but it fits the hard constraints and maybe the mutations will yield better results. The F_{wert} function will evaluate this to -110 which is very bad. Our search should mutate this as a consequence. A mutation will swap two people randomly and then evaluate the result. An example is as follows.



Here p_{1tl} has swapped with p_{2m} and has resulted in a slightly better fact with a worth of

-70. This mutation occurs many times and results in a large collection of facts that are evaluated by f_{wert}. This process is shown below.



This is only a small look at some transitions but we can assume we ran out of time. These facts all result in worths of -70, -65, and -70 respectively from left to right. From this, f_{wert} will find the best solution which is the fact with the highest score namely the score -65. In this case f_{select} wasn't invoked but when a tie occurs f_{select} is called. The function halfkill would also be used if the state got too big.

Notes and Development Ideas

It may be possible to constantly maintain a max-heap of certain values from f_{WERT}. Doing so might make for a quick implementation of the search control as a whole, although it limits our choice of f_{SELECT}. Namely, it would have to be based on whichever of two equivalently worthy choices bubbled to the top of the heap. This is “random” in the sense that it is “arbitrary”, but not in the sense that it would behave differently each time the program was run. This is a problem, perhaps.

We may run into a problem of being able to find a “local maximum” specimen which has a higher f_{WERT} than any of the possible mutations of it. In such a case, we may wish to check it against our other specimens, and if it were not the best we had, it should be killed and a new random specimen generated. This would prevent stagnation and lack of progress. (This is, of course, assuming we continue without a crossover function).

The Or Tree Based Search at the beginning to create the initial state may be implemented in a less structured way, eventually, if it shows itself to be a time bottleneck. Not all the formalities of a true Or Tree Based Search as discussed in class are necessary for this purpose.

Towards the end of a search (let's say, with 20 facts in the states), we may implement a “panic mode” for the very end which kills of the 15 worst facts without replacing them and focusses on improving the remaining 5 for the remainder of the time, so that we can hopefully return the best possible answer. This is risky of course, we'll have to test how it actually works.

Or Tree Based Search

Alternative solutions form the core of Or-Tree search. We generate all children of the current node, decide on the most fit of the alternatives, and halt when we have found our most viable solution. We define the fitness of each offspring in terms of hard and soft constraint violation.

Search Model

As per Denzinger's definitons (Kremer, 2013), we can define the search model (A_V) of our or-tree in terms of State (S_V) and Transition Rules (T_V).

$$A_V = (S_V, T_V)$$

Search model can be considered in terms of various paths to solving our problem, while the transition rules can be taken in terms of how we branch or halt. We define these formally below.

Search State

As per Denzinger's definition (Kremer, 2013), we define the search state as a the division of our problem and its current room assignments. It will be represented by a list of persons and their assignments to rooms. Please refer to Kremer's (2013) notes for a formal definition of the *Otree* and its recursive structure. Our search state is, then, defined in terms of the decisions that we make on our way through the *Otree*.

$$S_V \subseteq Otree \mid isValidWalk(S_V, Otree)$$

Transition (Extension) Rules

Our transitions are defined by either solving the problem or producing alternatives to our current problem. We are able to branch from our current state if there are valid alternatives to our solution. As well, we can determine a problem as solved if there exists at least one alternative which contains a valid room assigning which violates no hard constraints. We can also determine that a problem is unsolveable if there is no possible assignment available that does not break a hard-constraint. These form our Erw_V, which then lets us define our transitions as Kremer (2013) does, below:

$$T_V = \{(s_1, s_2) \mid s_1, s_2 \in S_V \wedge Erw_V(s_1, s_2)\}$$

Search Process

Using Denzinger's definitions (Kremer, 2013), we can define the search process as follows.

$$P_V = (A_V, Env_V, K_V)$$

Our model (A_V) is defined in the previous section as above. As well, our Env_V is exactly the same as our Env_{SET} for the exact same reasons. Therefore, we shall not explore either further.

Our K_V is similar in concept to K_{SET}, but requires some subtle difference. First, we do not consider non-viable solutions. These are excluded by our alternative generating function, and therefore need not be assessed. Therefore, we are purely evaluating the soft constraint worth of each of our alternatives generated by *Altern*. As well, while difficult to write formally, we consider managers, team and project leaders before other persons such that they get the largest possible office. This should help make smart decisions throughout our walk of the *Otree*.

Search Instance

Using Denzinger's definitions (Kremer, 2013) we define the search instance of our problem formally as

$$Ins_V = (s_{0\text{-}V}, G_V)$$

where $s_{0\text{-}V}$ is our initial search state, and G_V is our goal function. We define our *Sisyphus I* initial search state and goal functions below.

Initial State

Our $s_{0\text{-}V}$ represents our empty assignment, whereby we have a list of all persons and no assignments to rooms. We represent this formally as follows:

$$s_{0\text{-}V} = (pr, ?) \mid pr \in Prob, isEmptyAssignment(pr)$$

Goal State

G_V is a function such that we return *yes* iff there is at least one solved alternative available to our current node. Similarly, our G_V will return *no* iff all alternatives are unsolveable or there are no alternatives available for our current node. We define this formally as follows.

$$G_V\text{: } Prob \rightarrow \{yes, no\}$$

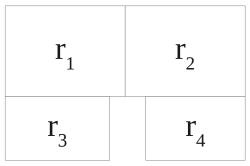
$$G_V(s) = \{ans \mid ans \in \{yes, no\} \bullet ans = yes \text{ iff } existsSolvedLeaf(s) \bullet ans = no \text{ iff } (noSolveableLeaves(s) \vee noAlternativesAvailable(s))\}$$

No further tailoring is required to fit this to our *Sisyphus I* problem, and therefore we will move onto our implementation example.

Example

This is a very simple example. The number of people ensures there is an ideal solution but also demonstrates the choices made to come to a good solution

6 People	*4 Rooms*
1 Manager	2 Large
1 Team Lead	2 Small
2 of Manager's team	
2 of Team Lead's Team	



This example will start with a set of unassigned people as the start state. The children will consist of various additions of people to rooms. There is a set of rooms in the environment and an assignment of one person to a room will look like p → r such that $p \in P, r \in R$.

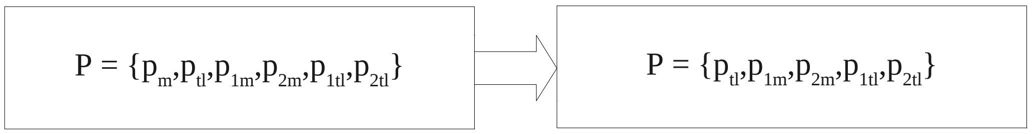
$P = \{p_m, p_{tl}, p_{1m}, p_{2m}, p_{1tl}, p_{2tl}\}$ $R = \{r_1, r_2, r_3, r_4\}$

Note: p_m is the manager, p_{tl} is the team leader, p_{1m} and p_{2m} are the manager's team members, and p_{1tl} and p_{2tl} are the team leader's team members. R is the set of rooms where r_1 and r_2 are large while r_3 and r_4 are small.

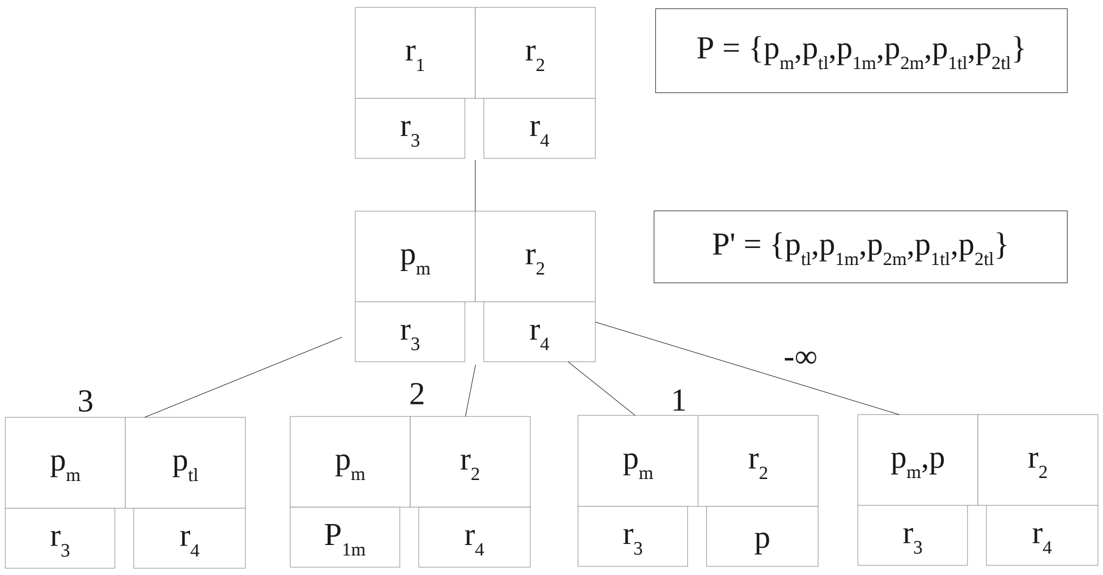
Now there are 24 children of the initial state which are the possible movements of a single person to a room. This can be seen from each person having four possible rooms and six different people thus 24 possible allocations.

We can look at each person and evaluate their choice's worth according to f_{altern}. If we evaluate p_m and p_{tl} first, the assignments $p_m \rightarrow r_1$, $p_m \rightarrow r_2$, $p_{tl} \rightarrow r_1$, and $p_{tl} \rightarrow r_2$ are all of equal but highest worth. This is because the soft constraints have a higher penalty for a manager not being in a large room. Assigning the people who are not managers to a room at this time is of less worth but equal for every person. (Possible priority for smaller rooms to allow room for managers, although the manager should be allocated first.)

Due to our control, the choice will be among $p_m \rightarrow r_1$, $p_m \rightarrow r_2$, $p_{tl} \rightarrow r_1$, and $p_{tl} \rightarrow r_2$ these are all equal so a random choice is made namely $p_m \rightarrow r_1$.


P = {p_m,p_{tl},p_{1m},p_{2m},p_{1tl},p_{2tl}}
P = {p_{tl},p_{1m},p_{2m},p_{1tl},p_{2tl}}

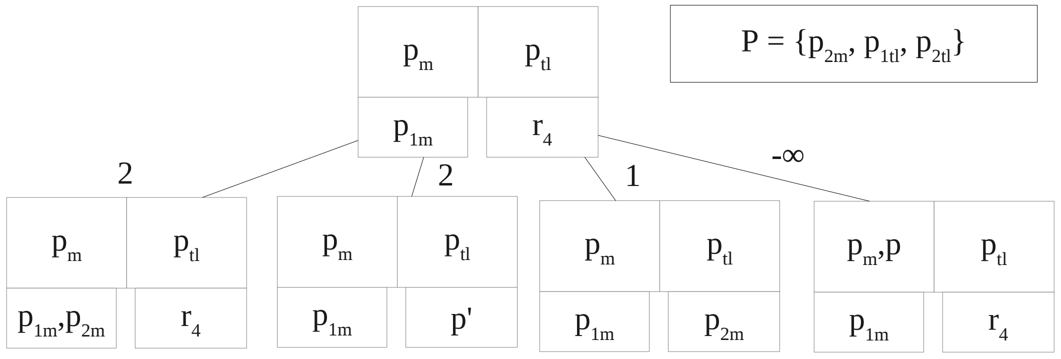
Our new current state is one without p_m and so we are left with 5 people to place. Now that p_m is placed, a hard constraint can be violated when $p \rightarrow r_1$ where $p \in P$. These choices are given a worth of negative infinity by f_{altern} so they can't be chosen. The assignment $p_{tl} \rightarrow r_2$ is a higher worth choice for the same reason as $p_m \rightarrow r_1$. The next highest priority movement at this step would be the manager's team members being placed in r_3 adjacent to r_1. The least worthy choices are going to be p_{1tl} and p_{2tl}'s placement. This is illustrated below.



Note: $p \in P'$

This visualization of our choices is paraphrased but we can see that $p_{tl} \rightarrow r_2$ is the best choice. When this path has been chosen, only the 4 team members will be left. Once again f_{altern} will evaluate the worth of these choices. The hard constraint violations arise from any transition $p \rightarrow r_1$ or r_2 where $p \in P$. The best choice in this case will be allocating people to adjacent rooms to their leader or manager. Thus the worth of $p_{1m} \rightarrow r_3$, $p_{2m} \rightarrow r_{r3}$, $p_{1tl} \rightarrow r_4$ and $p_{2tl} \rightarrow r_4$ are all equal and the best choices. Since they are all equal f_{altern} will choose randomly.

We can assume $p_{1m} \rightarrow r_3$ is chosen. At this point the choices are more limited. We can see the choices completely below.



Note: $p' \in P - p_{2m}$ and $p \in P$.

The last transitions of the search are going to be very similar. The best transitions are the ones that put the person closest to their leader. If we go down $p_{2m} \rightarrow r_3$ we have only one place to put the remaining people that is viable. This example has demonstrated the strong control and the success this method can have in searching for a solution.

Notes and Development Ideas

While in theory there is no backtracking for an Or-Tree based search, it is not guaranteed that there will always be a solution for any given path we choose. Therefore, if all children are unsolvable for a given state, then we need to practically choose an alternative route. This isn't likely to happen in our incarnation of the Sisyphus I problem. Therefore, I would suggest that we ignore it for now. Rather, we want to define a strong control function to avoid backtracking altogether.

Another possibility for an unsolvable path is to create an alternate “hero” function that messes with the entire tree to try and turn it into a valid path again.

Correspondence with Kremer gave us the idea to implement an “iterator” that pretends to generate the tree but doesn't actually allocate much memory. This seems like a really good idea so we're looking into it.

The idea that multi-threading could allow us to try multiple paths at the same time has been discussed but rejected for potentially reducing the amount of work done on either. Our control has to be trustworthy to always take the best seeming choice possible then, however.

References

Kremer, R. (2013). Search Paradigms: Or Tree Search. In *CPSC 433: Artificial Intelligence Course Material*. University of Calgary. Found at URL: http://kremer.cpsc.ucalgary.ca/courses/cpsc433/W2013/index.html#textbooks

Kremer, R. (2013). Search Paradigms: Set Based Search. In *CPSC 433: Artificial Intelligence Course Material*. University of Calgary. Found at URL: http://kremer.cpsc.ucalgary.ca/courses/cpsc433/W2013/index.html#textbooks